1982

Software engineering practices: their impact on the design of a program maintenance manual.

Teuscher, James Howard.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SOFTWARE ENGINEERING PRACTICES:
THEIR IMPACT ON THE DESIGN OF A
PROGRAM MAINTENANCE MANUAL

by

James Howard Teuscher
December, 1982

Thesis Advisor: Ron Modes

Approved for Public Release; Distribution Unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Software Engineering Practices: Their Impact on the Design of a Program Maintenance Manual	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis December, 1982
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): James H. Teuscher	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School Monterey, California 93940	12. REPORT DATE: December, 1982
	13. NUMBER OF PAGES: 84
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Engineering, Software Documentation, Maintenance Manual, Structured Programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The cost of software is fast becoming a major slice of DOD's automated data processing budget. Most of this cost is directly related to the maintenance of existing software. A primary cause is poor or non-existent documentation which leads to high costs when it comes time to change the software to correct errors, add enhancements, or to comply with changes in Federal regulations/ DOD policies. (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 68 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued) Block # 20

This thesis looks at the various software engineering techniques available to programmers and managers for the development of software documentation. A set of guidelines for an "ideal" program maintenance manual is proposed. These guidelines are based on current DOD standards, examples of software maintenance manuals from industry, and applications of current software engineering practices.

Approved for public release; distribution unlimited.

Software Engineering Practices: Their Impact on the Design of a Program Maintenance Manual

by

James Howard Teuscher
Lieutenant, United States Navy
B.A., State University of New York, College at Oswego, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
December 1982

ABSTRACT

The cost of software is fast becoming a major slice of DOD's automated data processing budget. Most of this cost is directly related to the maintenance of existing software. A primary cause is poor or non-existant documentation which leads to high costs when it comes time to change the software to correct errors, add enhancements, or to comply with changes in Federal regulations/DOD policies.

This thesis looks at the various software engineering techniques available to programmers and managers for the development of software documentation. A set of guidelines for an "ideal" program maintenance manual is proposed. These guidelines are based on current DoD standards, examples of software maintenance manuals from industry, and applications of current software engineering practices.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. THE NEED FOR DOCUMENTATION

A recent Govenment Accounting Office report, reviewing computer software maintenance in Federal ADP agencies, produced some interesting observations [Ref. 1].

1. Two thirds of the programmers at 15 Federal ADP agencies spent their time on software maintenance.
2. The Department of Defense will spend approximately $3 billion in 1981 on software.
3. Agencies have made little effort to effectively manage and minimize the resources required to maintain computer software.
4. Software is often maintained by people who did not develop it.
5. Poor documentation often results in rebuilding an entire system of programs because understanding and modifying an existing program may be more trouble and expense then building a new one.

The Federal Government is not alone in its software maintenance 'crises'. All ADP users, including private industry, are being swallowed up by the "tar-pits of software management" as Fred Brooks has so vividly described [Ref. 2]. The costs to the government and private industry, in terms of dollars and manpower, is increasing at an alarming rate. Costs for software (procurement and maintenance) are expected to reach $200 billion by 1985. DOD estimates it will spend $30 billion by 1990 for embedded software alone [Ref. 3,4].

A large share of these costs are for software maintenance (see Figure 1.1). Various studies have shown that from forty to ninety-five percent of manpower effort in most ADP organizations is spent on software maintenance [Ref. 5-14].

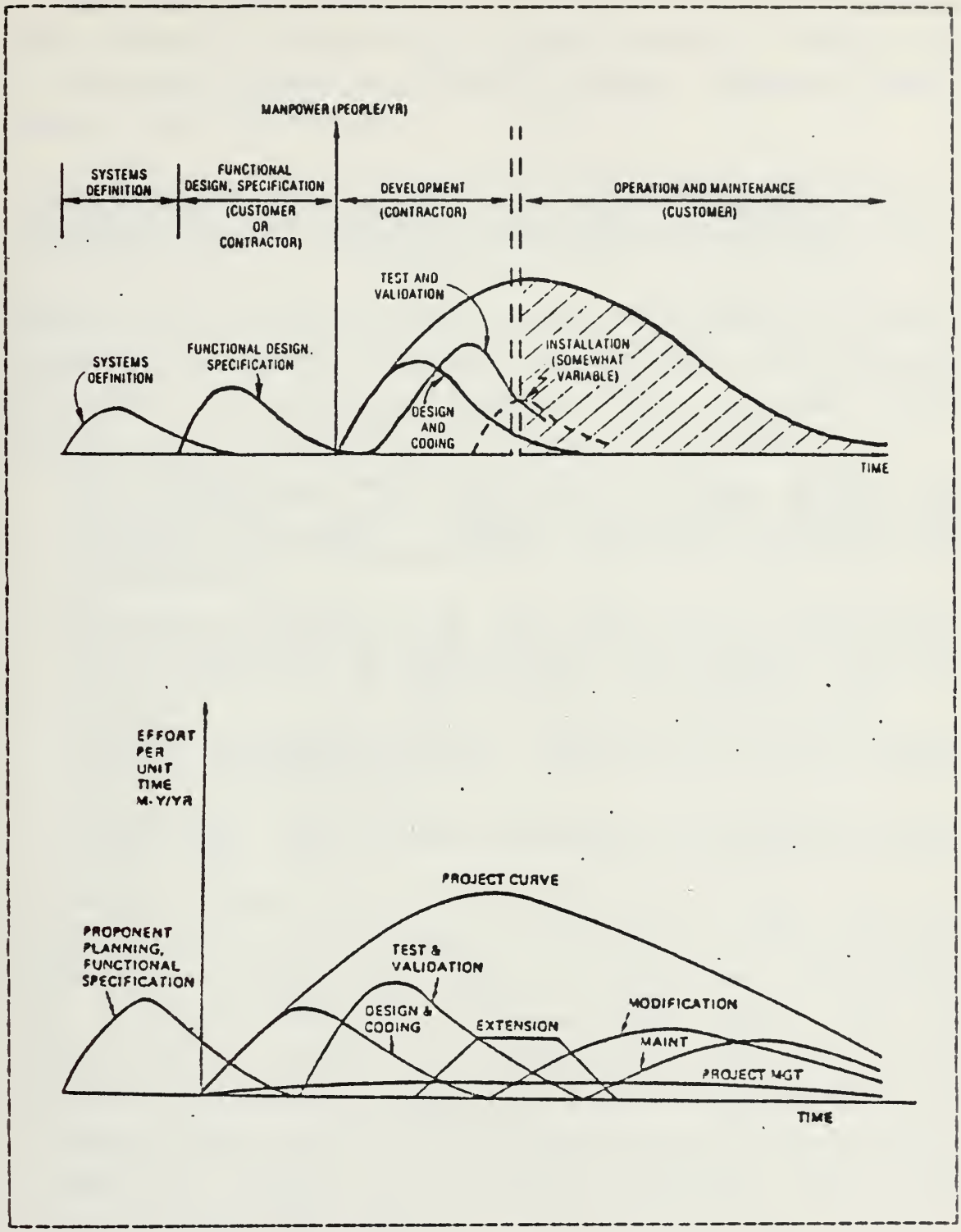

Figure 1.1 Manpower Loading and Maintenance costs.

There are many reasons for the wide variation in effort being devoted to software maintenance, however, a definition of maintenance should be presented first. Software maintenance is best defined as:

> That activity which is concerned with making changes to software for the purpose of improving or correcting the software without introducing new errors [Ref. 15].

Despite the fact that large amounts of manpower and other resources are spent on software maintenance, few managers comprehend the underlying reasons. There are four basic issues behind the problems of software maintenance:

- Maintenance is considered less glamorous, interesting and challanging as compared to system design and programming; hence, there is little incentive for computer personnel to become involved in maintenance activities (or document their time and effort spent on maintenance).
- During development it is often too early in the project to forsee problems which may occur during maintenance; as a consequence maintainability (the property of software which makes maintenance activities easy to perform) is not provided for in the system design.
- Project management does not always recognize that maintainability considerations should be an integral part of the design process.
- It is very difficult to modify or simplify the structure of a program after the program has been written. [Ref. 15]

Weapon system software and real-time system software used extensively by the Department of Defense (DOD) suffers from similar problems [Ref. 16]. De Roze [Ref. 17] reports that Air Force avionics software costs about $75 per instruction to develop, but maintenance costs are around $4,000 per instruction. SAGE, a military defense system, had an average annual cost of 20 million dollars after ten years of operation compared to an original development cost of 250 million dollars [Ref. 18].

The need for efficient, cost effective software maintenance is important because "....of the need to keep DOD real-time weapon system software operating as error free as possible and the need to check the escalating cost associated with modifying this software" [Ref. 16].

Good documentation is seen as one of the best tools for improving software maintenance [Ref. 19]. In general, documentation serves as a means of communication within an organization, especially over time. Documentation facilitates the training of new personnel and aids in modification of a software system by people other than the original development programmers. Silbey [Ref. 20] sees documentation as addressing three primary groups of people; managers, programmers/system analysts, and users. He also argues that the software documentation must be clear, comprehensive, detailed and well structured in order to be used effectively. Good documentation can and must provide a great deal of useful information. Specifically, this information has to communicate to the maintenance personnel enough detail so that enhancements and changes to the software can be easily made and do not produce unwanted effects in other parts of the system [Ref. 21]. Table I lists examples of the types of information needed in documentation.

Documentation has several important uses beyond a general description of code. During the software development phase it is used for communication between members of the design team, for training new personnel assigned to the project and as a basis for design reviews. During the software maintenance period the documentation serves again as a training base, a guide for modification and error checking, as an organized collection of design information, and as a historical trace of the software's production and operation.

> We wish to emphasize the necessity of considering the generation of timely documentation as an integral portion of the software development process [Ref. 22].

TABLE I

Information Documentation Provides

-A historical record of software system development.

-A detailed analysis of software system design.

-A well structured source code listing with comments on logic and processing flows.

-Identification, logical, and physical characteristics of the Data Base.

- Requirements, operating environment, and design characteristics of the system.

-Written instructions for non-ADP personnel that explains what the system (software and hardware) does and how to use it.

-Format of input data and output reports.

-Technical information about data collection requirements.

-Detailed specifications, descriptions, and procedures for all tests and test data.

Documentation is an important product of sound software engineering design rather than a simple by-product of design. Documentation has to be clear and concise. The documentation format has to be convenient and simple to use. The documentation has to be organized in a hierchical fashion in the same manner that the code is structured. All design decisions and their impact has to be explained, and the methodology for modifications decided in advance. Version control, and the accounting methods for modification to the software, has to be thorough [Ref. 23].

How much and what types of documentation must be decided during the the design phase of system development. Much too often these decisions have been put off until late in the system development cycle which results in poor or non-existant documentation [Ref. 1].

If we are to improve the maintainability of large software systems we must also "design for change" as Parnas suggests [Ref. 24]. This includes designing good documentation to tell us what a software system does and how it does it.

B. PURPOSE AND ORGANIZATION OF THESIS

The purpose of this thesis is two-fold. First it will examine and review Federal and Department of Defense directives on standards for documentation. Included will be a summary of the various techniques of documentation from the technical literature. Second, a design for a Programmer's Maintenance Manual will be presented which incorporates the latest concepts in software engineering. The reasons behind a particular design and the benefits to be gained will be discussed.

Chapter II of the thesis will review the concept of the software life cycle and how software maintenance activities relate to different life cycle phases. A discussion of current DOD directives in software management that govern weapon system software will be given here. Techniques currently available for programmers and software contractors for documentation will be described in Chapter III. These techniques include methods for representing program logic such as structured programming, data structures and commented program listings.

Chapter IV contains a description of a Programmer's Maintenance Manual based on DOD requirements for software documentation and recommendations for changes to such a manual. The manual is designed from the point of view that the program listing now provides a great deal of "self-documentation" resulting from advances in structured programming techniques. In addition the purposes and goals of a manual in general are presented with a view that any manual should be designed for its intended user.

Finally, Chapter V will provide conclusions and recommendations. An appendix contains a copy of the current DoD standard for a program maintenance manual.

II. BACKGROUND OF SOFTWARE DOCUMENTATION

A. THE SOFTWARE LIFE-CYCLE

All software, tactical weapon system and general data processing, must go through several phases or steps from the time a need for a software system is conceived to the point where the system is operational. This series of steps is generally referred to as the software life cycle. The different phases of the life cycle have various names when discussed in the literature. Many representations of the life cycle are described. Figure 2.1 depicts one such model based on the well understood DOD system life cycle as presented in DOD Instruction 5000.1. Figure 2.2 represents a more general approach to the life cycle phases.

1. Software Development

In order to understand better what must be done to create a software system, an informal description of each phase is provided [Ref. 25].

a. System Feasibility and Analysis Phase

The eventual user of a system discovers a need for which a computer based information system or weapon system seems to be the answer. The nature of the need is analyzed; the outlines of the type of system that would satisfy these needs are established. The most feasible and superior concept is defined.

DEFENSE SYSTEM LIFE CYCLE MAJOR PHASE	SOFTWARE LIFE CYCLE SUBPHASE
Conceptual	Requirements Definition
	Requirements Validation
Validation	Validation
Full-Scale Development	Full-Scale Development
Production	Production
Deployment	Debugging
	Fine Tuning
Support	Maintenance
	Modification

Figure 2.1 DOD Software Life Cycle Model.

b. Requirements Specification Phase

Functional descriptions of the system are developed using a particular formal methodology. Constraints on the system's structure and resource usage are identified. Economic constraints on the development process itself are stated. This phase may be an iterative process that oscillates between the statement of specifications and refinement of the design. Documentation standards and management objectives should be defined and listed. This phase is to clearly and concisely state what the system is to do - not how to do it.

c. Product Design Phase

Working from the specifications the overall hardware/software architecture is conceived. The underlying structure of the problem to which this system will be a solution is sought out. When this structure becomes clear, a

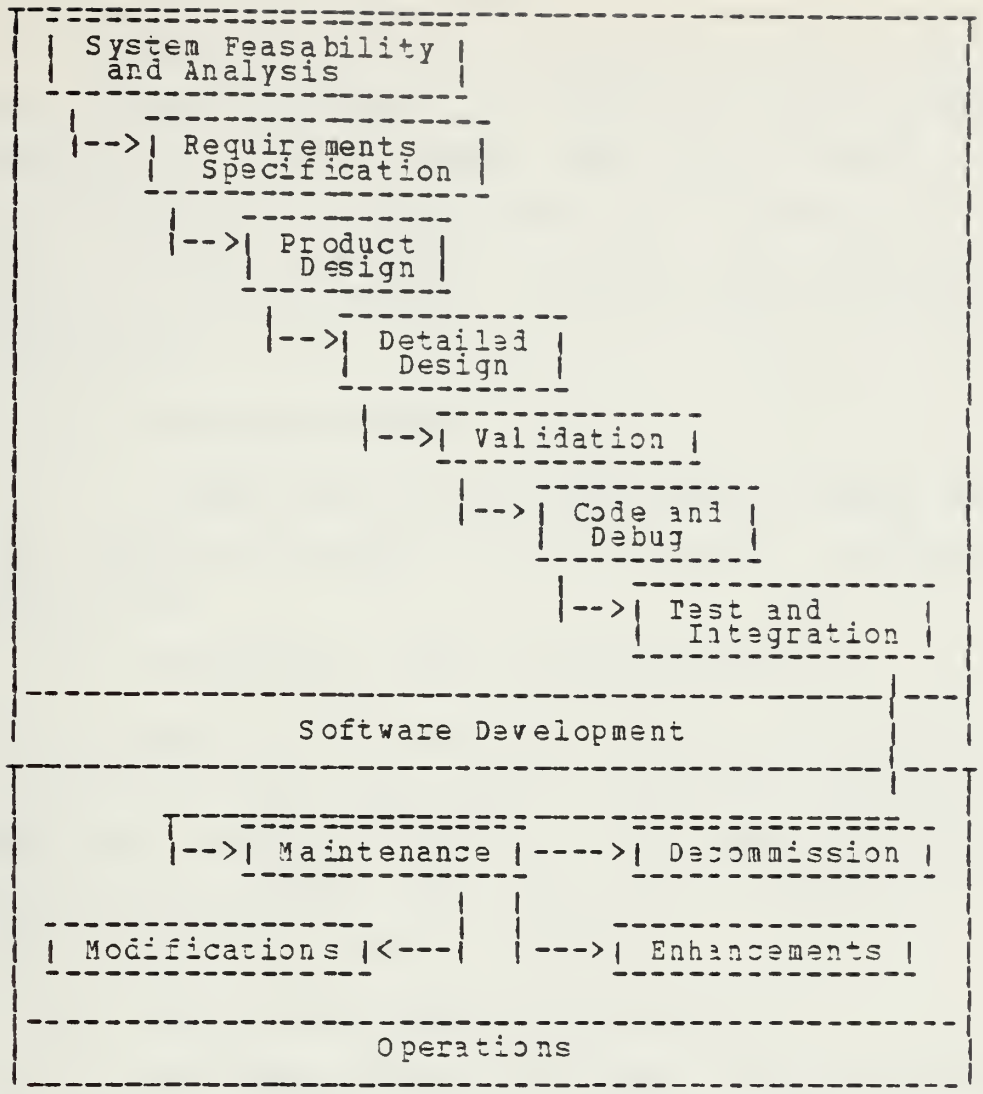

Figure 2.2 Software Life Cycle--General Schematic.

design of the system is devised. The design is necessarily at a gross abstract level of detail. The parts of the system and their relationships, the basic algorithms that the system will use, and the major data representations that will be needed are created. During this phase the basic test plan is developed, preferrably by an independent design team.

d. Detailed Design Phase

The major parts of the design are now refined in detail. Precise algorithms and data structures are defined and spelled out. If not already done in the Product Design Phase, decisions as to which parts of the system will be in software and which in hardware are made. Both detail design and product design may require several levels of refinement and reiteration.

e. Validation Phase

At this point in the development a check must be made to ensure the design, in all details, fulfills the specifications.

f. Programming, Code and Debug Phase

Encoding of algorithms and data representations is accomplished in this phase. Individual modules are prepared and tested individually. All basic debugging is completed where possible. Some bugs may not appear until all the system parts are executed together.

g. Integration/System Testing Phase

All coded modules are placed together along with a sample data base. This produces a physical realization of the design. Integration of all the parts, system testing according to the system test plan, and performance evaluation is accomplished. In many cases a good deal of redesign and reimplementation may take place at this time to force the actual system to conform to the specifications and initial requirements.

2. Software Operation

Everything that happens after the software system is finished, delivered and finally accepted by the user falls into the operational half of the life cycle.

a. Maintenance Phase

Tauseworthe offers a definition of maintenance as "alterations to the software during the post delivery period that do not require a reinitiation of the software development cycle" [Ref. 26]. He also views the maintenance phase as any software related activity, principally supportive in form, which keeps the software system operational within its functional specifications. The main activity of maintenance programers is corrective in nature, commonly referred to as debugging. Tauseworthe considers enhancements as essentially changes to the specifications which enable the software to perform either a new task or a different but similar task. In each case the functional scope of the program changes.

Swanson [Ref. 27] distinguishes between three types of software maintenance; corrective maintenance, adaptive maintenance and perfective maintenance. Corrective maintenance is performed in response to failures such as the abnormal termination of a program or the failure to meet performance criteria. Adaptive maintenance is performed in response to changes in environments such as the installation of a new generation of system hardware. Perfective maintenance is performed to make the program a more perfect design implementation. For example to improve processing efficiency or to add new features.

Other authors feel maintenance has only two basic components; modifications and enhancements. Modifications are any changes to existing functions to

correct bugs and meet specifications. Enhancements are additions of new functions that were in the original design but never implemented or were added as a result of an iteration of the development cycle [Ref. 25]. Modifications and enhancements will be the terms used in this thesis to refer to software maintenance.

A more accurate and definitive model of the maintenance phase of the software life cycle and of the life cycle itself has been proposed by the Rome Air Development Center [Ref. 28]. Figure 2.3 illustrates two important items concerning this model. The first is that the process of software development is highly interactive (indicated by feedback arrows) in order to incorporate new requirements and changes to software specifications. Secondly, and more important, it emphasizes the importance of the maintenance phase. The model indicates that maintenance phases are to go through the same iterative steps shown for software development, that is: analysis, specification, design, coding, validation, test and integration.

b. Decommission Phase

During this phase an assesment of the software is made to determine its further use, or to be replaced in its entirety. This may be due to completely new requirements where it is considered more economical to replace the software than to modify the existing system or where procurement of new hardware dictates a new software system be developed.

B. SOFTWARE MANAGEMENT

Software management is critical to the successful operation of a large software system. The decisions made by managers of weapon system software projects will mean the difference between whether the final product is maintainable

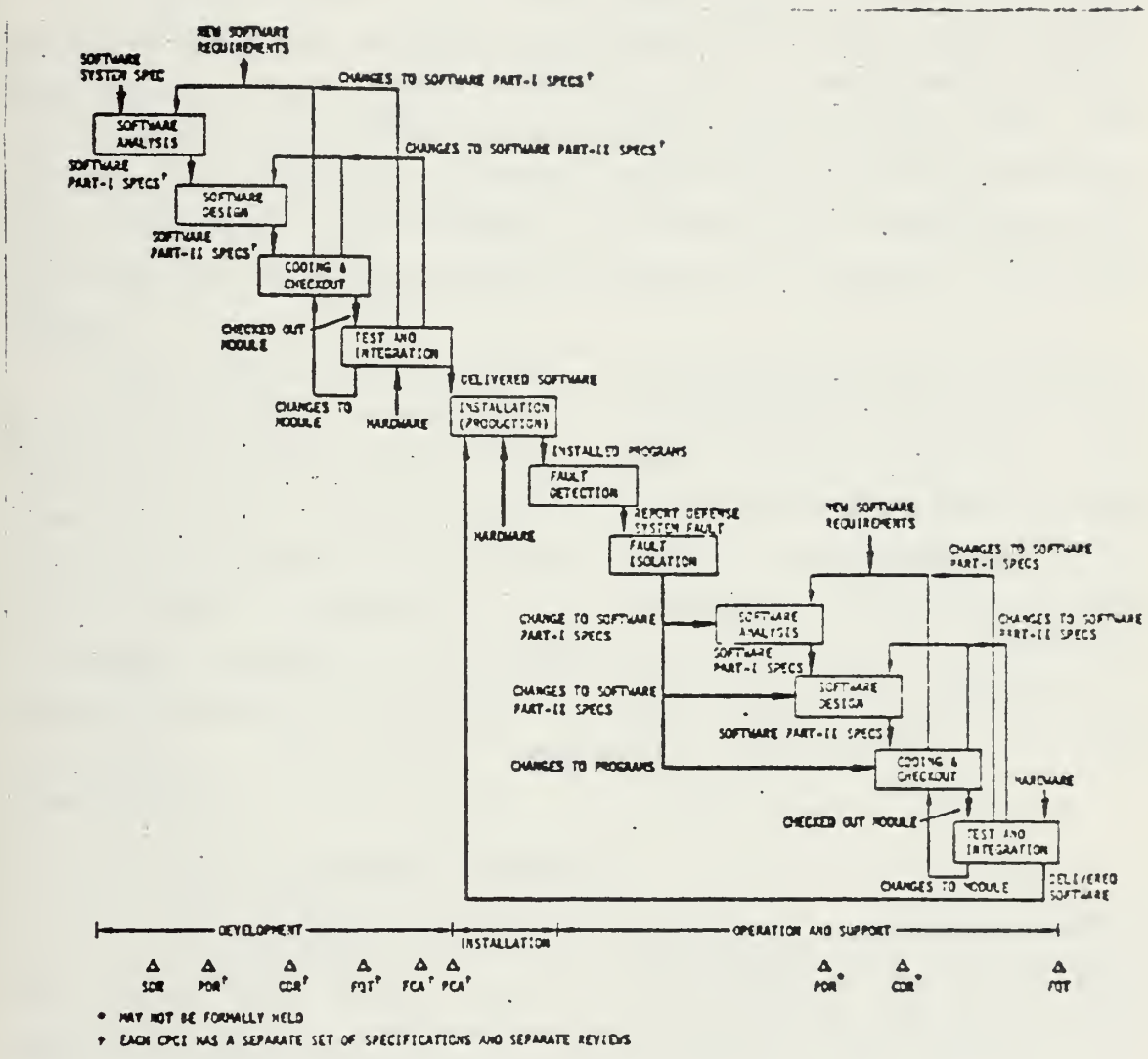

Figure 2.3 RADC Software Life Cycle.

or non-maintainable. Cave [Ref. 29] believes that: "Project failures are generally the result of improper or inexperienced management and not the lack of technical ability." Cave concludes that the successful development of large software systems can be achieved in a consistant manner.

In Cooper's point of view [Ref. 30] a common stumbling block of software project management has been that management would always seek to optimize the development process in trying to meet budget and schedule constraints. This type of approach creates an initial design with little documentation resulting in increased difficulty in maintaining the software and a corresponding increase in overall life cycle costs.

While there is no step by step process which will guarentee successful development of maintainable software, some general policies may be stated that are quite helpful. Daly [Ref. 31] lists several items that have proved useful based on his experience in managing software developments. Table II provides a listing of two different approaches. Daley recommends Method 1 in order to produce a cost-effective, more maintainable software product. He also recommends the application of strict management objectives to guide development.

1. DOD Management Policies

In December 1974 a DOD Software Steering committee was established with a goal of identifying critical weapon system software problems and to recommend policies for their solution.

To support this committee the MITRE corporation, in conjunction with John Hopkins University [Ref. 32,33] conducted a study of weapon system software management. It was concluded that "...the major contibuting factor to weapon system problems is the lack of discipline and

TABLE II

Software Design Methods

Method 1	Method 2
High level language	Assembly language
Structured Code	Tight Complex Code
Composite design(hierarchy of small segments)	Large blobs of code
Parallel, top-down, bottom -up design all optimally used	Bottom-up design
Simple data structures and work areas (not) tightly packed	Tight, effecient, data structures and work areas (every bit used, no data duplicated)
Team approach to design (egoless programming)	One program-One man concept
IBM's structured walk-through for reviewing detail design and code	No detailed technical review of design or codee
Three seperate teams,one design, one tests, one evaluates	Original coder tests, integrates and helps evaluate his program
Complete set of hierarchy charts, sequence charts data maps and narratives, well commented listing	Detailed flow charts and general narratives, no consistency listing with comments
Detailed test plans for all test phases	No formal test plans
Program maintained by 30% senior programmers	Program maintained by inexperienced programmers or technicians
Only commercial documentation generated during development	Extensive non-commercial technical memorandum generated and placed in library
Strict management objectives established to guide development	No management objectives

engineering rigor applied to weapon system acquisition activities."

The software management steering committee incorporated this and other ideas into a comprehensive plan to include policy, practices, procedures and technology initiatives [Ref. 34]. Parts of the plan are intended as supplements to principles stated in DOD Directives 5000.1 and 5000.2. The first maintenance management policy states that "Ease of maintenance and modification will be a major consideration of the initial design."

Two techniques are used to provide management control to weapon system software. These are design reviews and configuration management.

a. Design Reviews

MIL-STD-1521(USAF) annotates and describes the requirements for the following technical reviews and audits on computer programs:

- Systems Requirements Review (SRR)
- Systems Design Review (SDR)
- Preliminary Design Review (PDR)
- Critical Design Review (CDR)
- Functional Confirmation Audit (FCA)
- Physical Configuration Audit (PCA)
- Formal Qualification Review (FQR)

A software maintenance guidebook [Ref. 35] provides a supplement to MIL-STD-1521. It describes items to be taken into consideration in order to optimize the maintainability of software. For specific definitions on any of the above items one is referred to standard.

b. Configuration Management

Configuration management consists of configuration identification, control, status accounting and auditing.

As part of the proposed requirements assigned to contractors for the development of weapon system software, MIL-STD-1679, Weapon System Software Development, states:

> The contractor shall establish and implement the disciplines of configuration management; namely configuration identification, configuration control, and configuration status accounting. The contractor shall be cognizant of the requirement for long-term life cycle support of the weapon system software. The appropriate degree of configuration management shall be applied to ensure completely accurate correlation between descriptive documentation and the program in order to facilitate post-delivery maintenance by software support personnel.

Configuration identification involves specifically identifying and labeling the configuration items at selected baselines during the software life cycle. Baselines are reference points or plateaus in the development of a system; a baseline is formally defined at the end of each stage in the system life cycle. For example the functional baseline is typically the requirements specifications document that outlines, in terms both the buyer and developer can understand and agree to, exactly what the system will do. Configuration items are the individual entities that, together, define and describe the baseline. [Ref. 36]

Configuration control provides the means to manage changes to the (software) configuration items and involves three basic ingredients:

- Documentation (such as administrative forms and supporting technical and administrative material) for formally precipitating and defining a proposed change to a software system.
- An organizational body for formally evaluating and approving or disapproving a proposed change to a software system.

- Procedures for controlling the actual changes to a software system.

Software configuration status accounting provides the mechanism for maintaining a record of how the software evolved and where the software is at any current stage of implementation. Software configuration auditing provides a means to determine how well the software product matches its associated documentation. [Ref. 36]

DOD Directive 5000.29, Management of Computer Resources in Major Defense Systems, states:

> Defense system computer resources, including both computer hardware and computer software will be specified and treated as configuration items.

The primary objective of software configuration management is the effective management of a software system's life cycle and its evolving configuration. Configuration is the final form, arrangement or design of the software [Ref. 36]. The importance of configuration management is that it gives one the ability to manage change during the development process. If a program maintenance manual is being designed in conjunction with and as a part of the software system development, then it should be placed under the discipline of configuration management. Changes in the design and contents of the maintenance manual can be matched or directly linked to changes in the software system. This is the intent of DOD Directive 5000.29.

2. DOD Directives and Standards

Policies and procedures for acquisition of weapon system software are different in most respects from those used for procurring automated data processing equipment (ADPE). The distinction made between these two catagories of automated systems is a direct result of the 1965 "Brooks Act" (Public Law 89-306,40, U.S.C. 759).

The Office of Management an Buiget (OMB) and the General Services Administration (GSA) administer the Brooks Act guidelines. ADPE is controlled by this Act and falls under the cognizance of the Assistant Secretary of Defense(Controller). Weapon system software, on the other hand, is excluded from the provisions of this Act and fall under the jurisdiction of the Office of the Under Secretary of Defense for Research and Engineering.

There is no centralized source of guidence with respect to weapon system software maintenance for DOD orgainzations to follow. There are many directives, regulations, specifications and standards that impact on weapon system software to varying degrees. The most important ones are described here.

a. Weapons Standard WS 8506

WS 8506 is considered to be a comprehensive and very good specification for the documention of program development, particularly in view of its early publication date (1971). One of its strong points is:

> A strategy for making each level of documentation responsive to the next upper level (subprogram design under program design) which represents foresight in the use of top-down design prior to the time this term was in vogue [Ref. 15].

It does not include such programming techniques as structured programming, but this technique was not developed at the time of its publication.

Its weaknesses include a lack of change procedures, documentation standards for diagrams (such as Hierarchy Input/Output or HIPO) and regression testing [Ref. 15].

b. MIL-STD-483 (USAF)

MIL-STD-483 (USAF) "Configuration Management for Systems, Equipment, Munitions, and ComputerPrograms," 1 June 1971, defines the entire spectrum of activites associated with controlling changes (a critical need for maintenance work) to computer programs.

c. MIL-S-52779 (AD)

MIL-S-52779 (AD), "Software Quality Assurance Program Requirements," 5 April 1974, requires that a Quality Assurance Program (QAP) be implemented specifically for the development of computer programs and related documentation. Even though this standard is concerned with the development phase, it is important because it can directly affect the quality of software documentation.

d. SECNAVINST 3560.1

SECNAVINST 3560.1, "Department of the Navy Tactical Digital Systems Documentation Standards," 8 August 1974, identifies, names and describes that set of documents necessary to support both the development and maintenance of tactical software. A review of 3560.1 was conducted to determine how well this standard supports software maintenance activity [Ref. 37]. As noted by this review there are some positive and negative aspects to the standard. Some positive aspects include:

- Applicable document statements.
- Resource budgets (time,space).
- Numerous examples.
- Content check lists.
- Interface descriptions.
- Test coverage.
- Detailed Table of Contents for each specification.

The deficiencies of the standard include a lack of requirements for the subject of traceability, a need for increased emphasis on validation, and the use of inconsistant or non-defined terminology. The review indicates the standard seems more orientated towards software development then to softwre maintenance. The review also notes the standard's strong orientation towards the Navy's tactical data system. Scheidewind, [Ref. 37], recommends "...a more general orientation might be preferable to achieve a wider applicability to a variety of software systems."

e. DOD DIRECTIVE 5000.29

DOD DIRECTIVE 5000.29, "Management of Computer Resources in Major Defense Systems," 26 April 1976, establishes DOD policy for the management and control of computer resources during system acquisition. Maintainability of software is called out as a major consideration during the initial design. It also directs that support items required for cost effective maintenance be specified as deliverable items. Documentation is listed and defined as a support item.

f. MIL-STD-1521 (USAF)

MIL-STD-1521 (USAF), "Technical Reviews and Audits for System, Equipment and Computer Programs," 1 June 1976, prescribes the requirements for the conduct of technical reviews and audits in conjunction with the documents defined in MIL-STD-483. Direction is provided concerning the review and audit of computer program configuration items and their associated documentation. Each type of review or audit is described in an appendix to the standard and can serve as a basis for checking compliance with maintainability requirements. The documentation discussed here is related more toward system design reviews then towards the documentation of program listings.

g. DODINST 5000.31

DOD Instruction 5000.31, "Interim List of DOD Approved High Order Programming Languages (HOL)," 24 November 1976, specifies which HOL's are approved for use in conjunction with DOD Directive 5000.29. Although this instruction allows for certain exceptions, it attempts to reduce the proliferation of HOL's in defense systems by limiting new development to six approved languages: CMS-2, SPL-1, TACPOL, JOVIAL, COBOL, and FORTRAN. Its major impact is in the standardization of compilers and in preventing each DoD activity from developing its own unique programming language.

h. MIL-STD-1679 (NAVY)

MIL-STD-1679 (NAVY), "Weapon System Software Development," 1 December 1978, establishes uniform requirements for the development of weapon system software within DOD. Strict adherence to the provisions of this standard will help ensure that the tactical software so developed will be improved over current versions of tactical software. MIL-STD-1679 includes developments in programming technology, and management such as structured programming and design walkthroughs, which have occured since the release of WS 8506. It stipulates the use of high order languages and specifies the use of configuration management for correlating documentation with the program for maintenance purposes. [Ref. 15]

III. TECHNIQUES TO SUPPORT SOFTWARE DOCUMENTATION

A. STRUCTURED PROGRAMMING

There are several definitions and goals of structured programming. Some of these goals relate to the design and testing of large software systems. One particular goal of structured programming is to organize and discipline the program design and coding process in order to reduce logic type errors [Ref. 38]. It is generally accepted that the goals of structured programming include those of software engineering. In particular these goals are: modifiability, efficiency, reliability, and understandibility of the program code [Ref. 39].

What must be shown is how structured programming supports documentation. This is not easy to do because structured programming is not a universal and well defined concept. It is defined in many places, [Ref. 39,40,42], but not always consistantly. However, its essence is fairly well understood. It is the practice of of programming using a limited but sufficient set of control constructs Figures 3.1 and 3.2 illustrate the five basic control constructs as required by MIL-STD-1679.

Myers [Ref. 18] provides a list of seven basic elements of a structured program which should be applied to help reduce program complexity and promote clarity of thought by the programmer.

- The code is constructed from sequences of basic elements.
- Use of the GOTO statement is avoided whenever possible.
- The code is written in an acceptable style (e.g. use meaningful variable names, avoid statement lables, avoid language tricks).
- The code is properly indented on the listing so that breaks in execution sequence can be easily followed

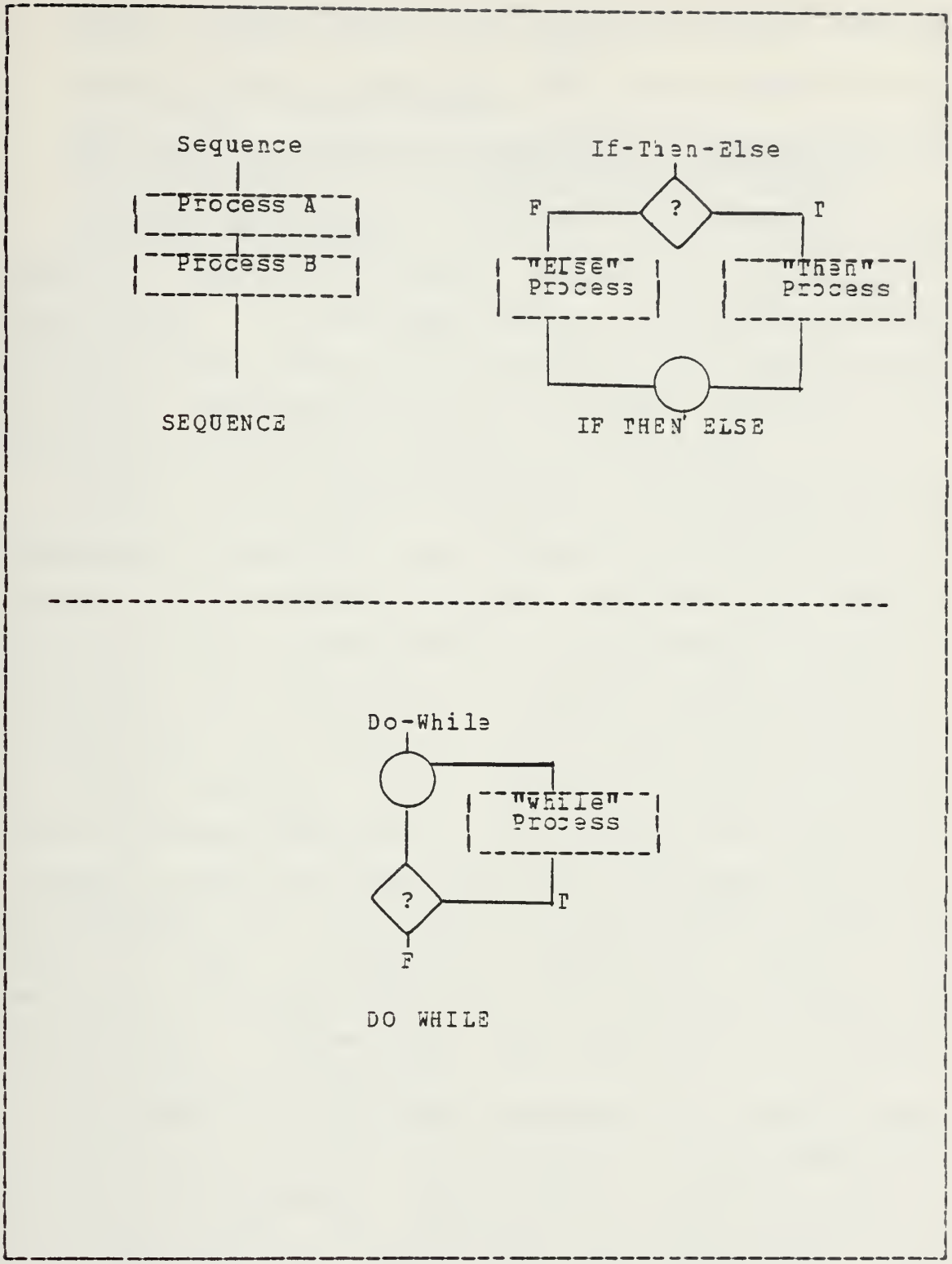

Figure 3.1 Basic Control Constructs.

(e.g. a DO statement can be easily matched with the ENDDO statement ending the loop).

- There is only one point of entry and one point of exit in the code for each module.
- The code is physically segmented on the listing to enhance readibility. The executable statements for a module should fit on a single page of the listing.
- The code represents a simple and straightforward solution to the problem.

Figure 3.3 provides an example of structured and unstructured coding. A structured program is structured in two different ways. First, it is structured with regard to flow of control and execution of the program. Second, it is physically structured by the use of indentation.

Another way of viewing structured programming is to see it as those attributes of a program that contribute to the readibilty of its form. For example consider the development of a file management system. Obviously one required function of such a system would be the capability to search for a given file identified by a specific name. Figure 3.4 illustrates such a function as coded in the recently developed DOD high order language ADA (see [Ref. 44] for details on the ADA language). As can be seen in Figure 3.4 structured programming makes for a more readable and discernible sequence of code. The main tenents of structured programming are displayed, that is; hierarchic relationships between the lines of code, a consistant indentation policy, and begin-end groupings which make it easier to follow the program flow. Comments are used extensivly to introduce each new module and structure.

The results of structured programming are readily apparent with easy to read code and easy to follow logic flow. Functions and procedures are confined to discrete areas in the program listing. The objective is to make the code, in a very meaningful and useful way, self-documenting, thus reducing the need for external documents describing the

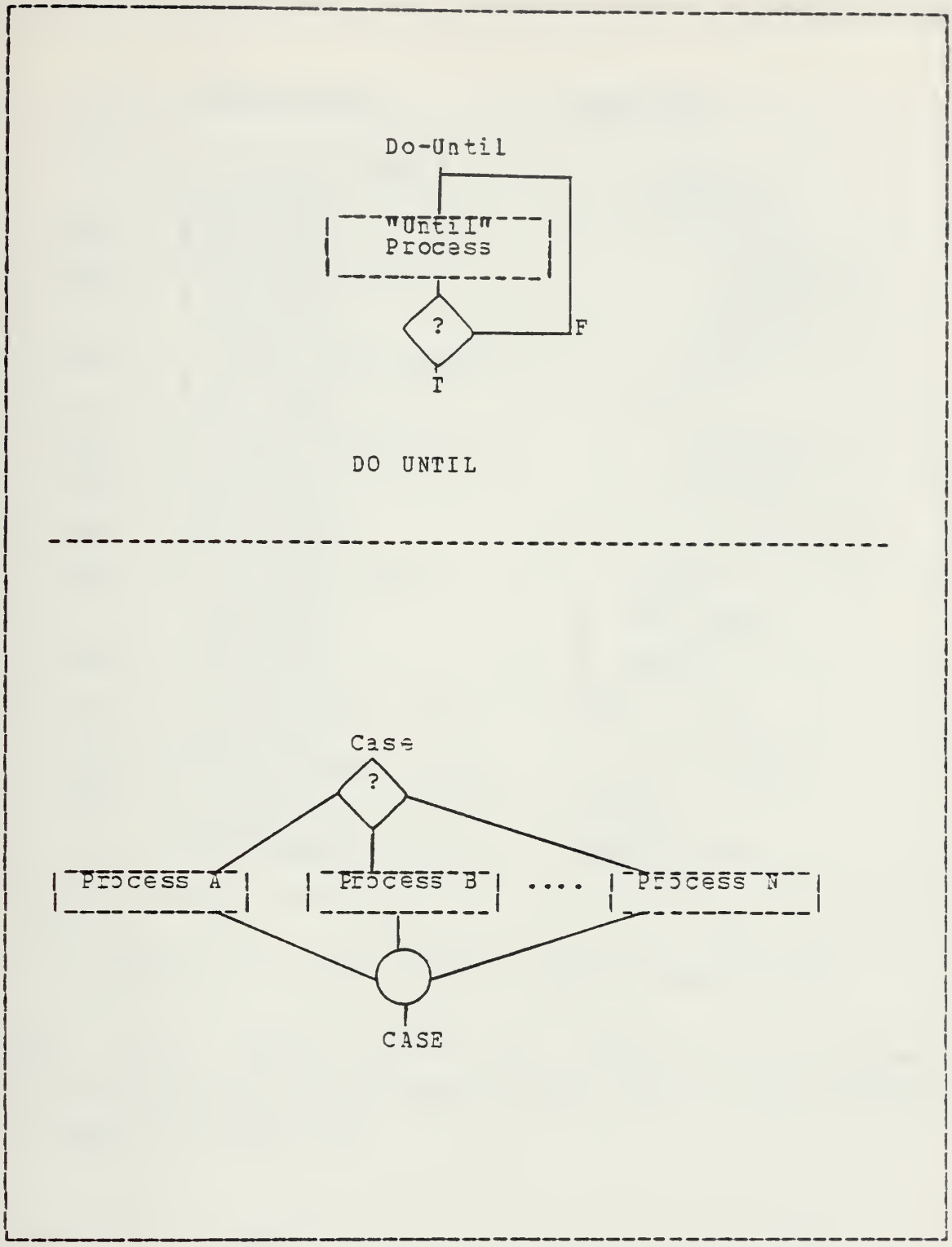

Figure 3.2 Basic Control Constructs.

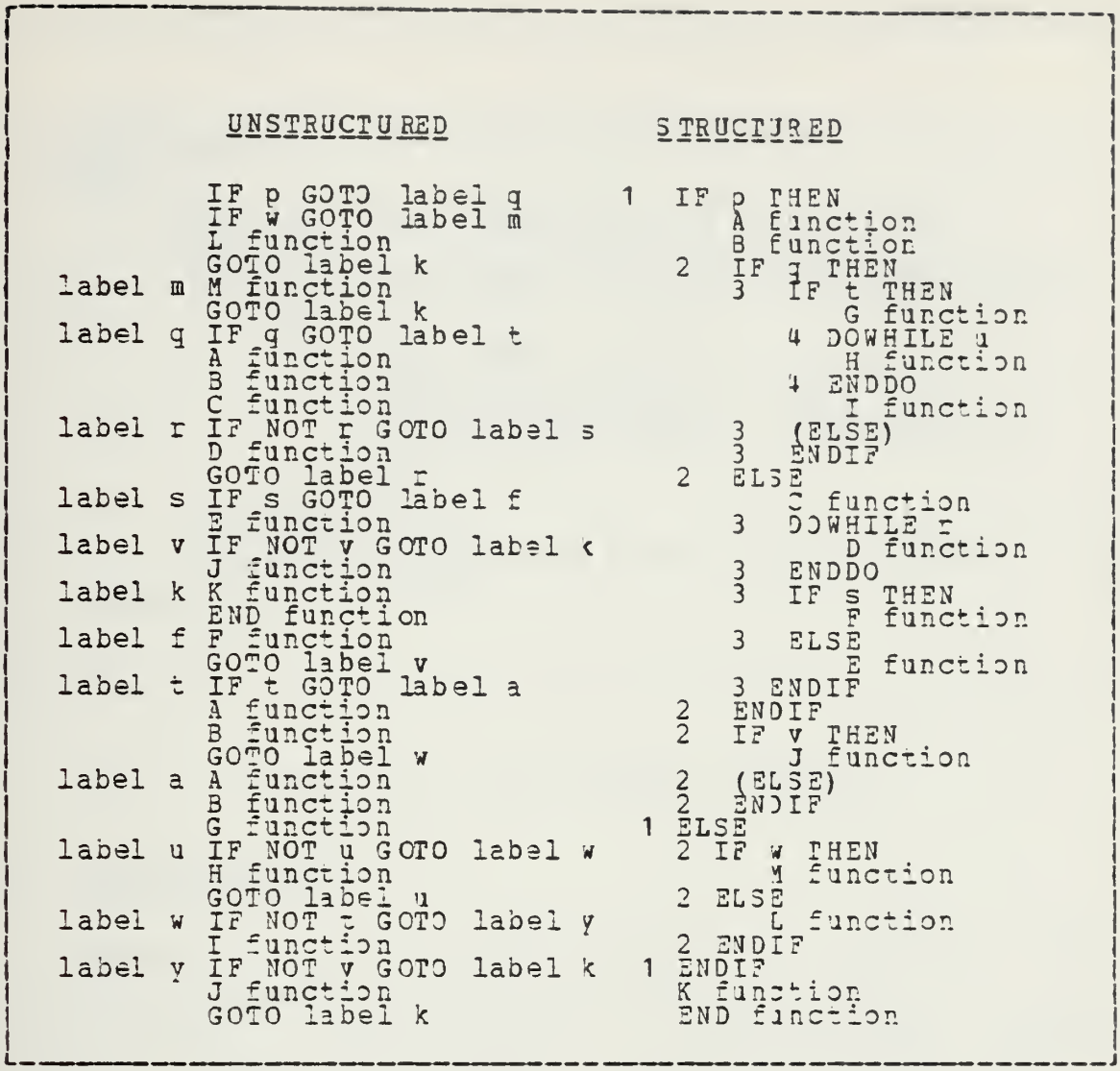

Figure 3.3 Structured vs. Unstructured Coding.

code. This has an effect on the design of the document called the Program Maintenance Manual and will be discussed in detail in Chapter IV.

```
function SEARCH (FILE_NAME;KEY_TYPE)return
FILE_INDEX is
begin
   -- Look for the specified Key record in the
   -- specified file, returning the index of its
   -- position
      for I in FILE_INDEX'FIRST..FILE_INDEX'LAST loop
      -- Search the whole data base, from first to
      -- last
            if FILE_MAP(I) /= NULL
            -- Check data base for a null or a
            -- match
               and then FILE_MAP(I) = FILE_NAME
               and then FILE_KEY(I) = KEY_TYPE then
               return (I);
            end if;
      end loop;
      raise BAD_FILE;
      -- Raise an exception if the desired record
      -- is not found
end SEARCH;
```

Figure 3.4 File Search Function.

B. MODULARIZATION

Modules are the building blocks of software. Good modular programming usually requires that external interfaces, such as input/output, be isolated into seperate modules. Modular programming itself is the practice of implementing software in small, functionally orientated pieces. These pieces are usually implemented as subroutines, functions or clusters of functions. Each module is devoted to one or more tasks related to a function; the module may be accessed from one or several places in the software system.

Modularity is often defined in terms of properties possessed by "modular" systems.

> A program is modular if it is written in many relatively independent parts or modules which have well defined

interfaces such that each module makes no assumptions about the operation of other modules except what is contained in the interface specifications.

Modularization consists of dividing a program into subprograms (modules) which can be compiled separately, but which have connections with other modules...A definition of "good" modularity must emphasize the requirement that modules be as disjoint as possible.

Modular programming is the organizing of a complete program into a number of small units...where there is a set of rules which controls the characteristics of these events.

Modularity deals with how the structure of an object can make the attainment of some purpose easier. Modularity is purposeful structuring [Ref. 38].

There is a trend towards defining modules in terms of the number of lines of code they contain. Programming standards frequently contain a somewhat negative approach such as "...modules shall contain less than X lines of code." Some authors feel that the size aspect is not as important to a module as its functional aspect. In other words a small complex module may be more difficult to understand than a large well structured module. However, the majority still favor limiting module size, when possible, to less than 100 lines of code in order to maintain modules as "discrete" entities and to aid comprehension [Ref. 39,40].

There are many advantages to using modules. Parnas [Ref. 24] argues that the most important aspect of modularization is the ability to anticipate and design for changes. If the function of a module changes, that module alone has to change. If the need for a function arises during the design phase, a module with this function can be invoked at the point of need. If an error in the program is found, the probability is that its correction will be limited to one module. Once a module has been tested, and can be compiled seperately, it can be reliably used in different places in the program. [Ref. 41]

Since modularization makes the structure of a program easier to understand while localizing functions and processes, the need for external documentation is again reduced.

As a final note, there has to be a distinction made between modualrity and structured programming.

> This distinction is necessary because program structure is a relatively meaningless concept in some programming languages, such as plain vanilla assembly language (as opposed to assembler enhanced with strong macro capability). In contrast, modularity is usable in all domains. Thus, the line between modularity and structure may be a tenuous one to walk, but it is an important one. Note that good modules may or may not possess good program structure and that bad modules may also possess good program structure. The two subjects are discernable [Ref. 43].

C. DATA STRUCTURE

In some programming languages there are at least two ways of working with a string of bits or characters. One way is to declare the string as part of a structure (Figure 3.5) and then manipulate the string by name:

```
OTHER_STRING = CHARACTER_STRING
```

```
Structure DATA_STRUCTURE; begin
     item CHARACTER_STRING 5 characters;
     item OTHER_VARIABLE 32 bits;
end Structure;
```

Figure 3.5 Example of a Data Structure.

A second method is to use a byte manipulating function to define the desired string at every point of usage in the program:

```
OTHER_STRING = BYTE(CHARACTER_STRING,5);
```

Suppose the character string named OTHER_STRING is utilized 100 times in the program. In order to change the string format using the first method (Figure 3.5), only one easily identified line of code has to change. In the second case there are 100 lines to change throughout the program.

Languages may or may not support such data structuring. COBOL and PL/I provide elaborate data structures for formatting report data for printout. In addition such changes as stripping off leading zeros, inserting a decimal point, adding check protect characters, and inserting a leading dollar sign are easily accomplished. At the other extreme FORTRAN and BASIC offer no data structure capability beyond the array [Ref. 43].

Data structure impacts the areas of software design and software maintenance. One methodology, called the Jackson method [Ref. 45], stresses designing the program structure by first looking at the data structures. According to Jackson, an algorithmic structure is highly related to data structure; programs that obey data structure design will have a more maintainable program structure as well. Obviously different application areas have different levels of need for data structure orientation.

One of the noticable effects of large software systems is that algorithms end up transforming one data structure into another data structure. This results in what Jackson terms a "structure clash." The reason for this occuring is commonly a proliferation of unnecessary code, lack of structured programming goals, or lack of design. Sometimes the "clash" is a result of maintenance, corrective and

enhancement, where the changes have added information or changed the data structure to meet the current need. Two solutions have been proposed in this area. Grouping or "clustering" of data and abstract data typing [Ref. 43, 49].

Where a set of data declarations interrelate, either by function or by context, they should be grouped together for ease of understanding and modification into one block. This is the "clustering" concept. If a change is needed it can be isolated into one block. Only those modules which use that block of data need be recompiled.

With the concept of abstract data typing, such as used by PASCAL and ADA, [Ref. 44] all data must be explicitly declared to be of a particular type. Certain types are defined in the language, but these must be supplemented by programmer-defined types. Associated with a type is a set of

```
type RECORD_FORMAT_1 is
     record
        EMPLOYEE_NAME: array(1..30) of CHARACTER;
        EMPLOYEE_RATE: float;
        EMPLOYEE_HOURS: integer range 0..99;
FILE1_RECORD: RECORD_FORMAT_1
```

Figure 3.6 Example of an ADA Data Structure.

legitimate values which objects of a type may assume and a set of operations which may be performed on that type. Figures 3.5 and 3.6 show examples of ADA data structures.

This kind of typing has three characteristics: (1) the properties of a type are defined centrally at the type declaration, and if they change they need only be changed in one place; (2) only the abstract properties of a type need

be known to reference data of that type, with implementation details "hidden" in the declaration; and (3) strong type matching may be enforced by the compiler, eliminating type mismatch errors as a reliability/maintainability issue [Ref. 43,44].

As a final consideration, it is always important to name data effectively. Meaningful names are always preferable. EMPLOYEE_NAME conveys far more information about the data value then EMPNAM. This supports the goal of readability and the ultimate goal of having the program be as self documenting as possible.

D. DATA COMMUNICATION

There are two basic types of data communication, intra-program and inter-program. Intra-program data communication is essentially communication between modules. This can be accomplished by parameter lists, global (or common) data and a recent concept called a data "cluster". A cluster is where a constrained "semiglobal" data base is shared among a limited number of processes or functions. [Ref. 42,43,44]

A parameter list identifies only those data items and formal parameters used by each individual module. The preferred method of intra-program communication is by the use of a parameter list [Ref. 43]. The main advantage gained is that all data used by the module are identified and isolated. It is not possible for the module to have a surprise effect on the code which invokes the module.

Global variables are used for large quantities of data when it is not convenient to specify such a large parameter list at each point of call. Most authors recommend minimizing the use of global data for good modular and structured programs. The reason for limited use is that a module may modify a global value whose original value was critical to

the calling environment. This may lead to undesirable and untracable side effects. [Ref. 40,42]

> A side effect is one brought about other than via a parameter mechanism. It is generally considered rather undesirable to write subprograms, especially functions, which have side effects. However, some side effects are beneficial. Any subprogram which performs input-output has a side effect on the file; a function delivering successive numbers of a sequence of random numbers only works because of its side effects; if one needs to count how many times a function is called then we use a side effect; and so on. Care must be taken when using functions with side effects that the function does not cause errors in other sections of the program. [Ref. 44]

The cluster concept is an attempt to fulfill the need for global variables. A data base and its family of manipulating procedures is isolated from the rest of the program. A program needing some or all of of the clustered data must explicitly import it. The cluster itself can distinguish between data and procedures which may be exported or not. The side effect problem is at least isolated and bounded. [Ref. 40,42]

Inter-program data communication is provided either by passing data flags and blocks through an operating system communication area, or by passing larger volumes of information on files. The UNIX concept of "pipes" in which predefined files are automatically passed from one program to the next is an example of this type of data communication. [Ref. 47]

The proper use of data communication clarifies program flow and tracability of processes. This, of course, supports the maintenance programmer's task in the correction of program errors and in his understanding of the program's documentation.

E. HIGH ORDER LANGUAGES (HOL'S)

High order languages are those computer languages which are essentially machine (hardware) independent. This is in contrast to assembly languages that were designed for a particular hardware architecture. For example, the INTEL 8080 assembly instruction set will not work at all on a Motorola 6800 based machine. In contrast a standard FORTRAN program can easily be used on any computer with an appropiate FORTRAN compiler.

High order languages tend to support the concepts just discussed (structured programming, modularity, data structure, etc.) to varying degrees. FORTRAN offers modularity, but is limited for program structuring and has almost no capabilty for data structuring. COBOL has excellent, robust capabilities for data structures. Other languages have their strong and weak points. [Ref. 43] Examples of good HOL code may be found in several references [Ref. 40,42].

Documentation is supported by High Order languages (HOL's) if the language is readable. FORTRAN is limited in this respect because of its six character limit on data and variable names. Some HOL's, APL for example, provide no readabilty at all. APL is very difficult to understand after it is written down, even by the person who designed the program.

ADA, the new programming language developed by the Department of Defense, promises to offer the best support for just about all the major software engineering principles. ADA is a large language since it addresses so many different issues. Some of the key goals of ADA are listed in Table III [Ref. 44].

The overall objective of the Department of Defense is to provide one standardized language for use in all tactical embedded systems. This can provide substantial economic

TABLE III

Key Goals of ADA

READABILITY

It is recognized that professional programs are read more often then they are written. It is important, therefore, to avoid an over-terse notation such as APL.

STRONG TYPING

This insures that each object has a clearly defined set of values and prevents confusion between logically distinct concepts. As a consequence many errors are detected by the compiler which in other languages would have led to an executable but incorrect program.

PROGRAMMING-IN-THE-LARGE

Mechanisms for encapsulation, seperate compilation, and library management are necessary for the writing of portable and maintainable programs of any size.

EXCEPTION HANDLING

It is a fact of life that programs of consequence are rarely correct. It is necessary to provide a means whereby a program can be constructed in a layered and partitioned way so that consequences of errors in one part can be contained.

DATA ABSTRACTION

Extra portability and maintainability can be obtained if the details of the representation of data can be kept seperate from the specification of the logical operations on the data.

TASKING

For many applications it is important that the program be conceived as a series of parallel activities rather than just as a single sequence of actions. Building appropiate facilities into a language other than adding them on via calls to an operating system gives better portability and reliability.

GENERIC UNITS

In many cases the logic of part of a program is independent of the types of values being manipulated. A mechanism is therefore necessary for the creation of related pieces of program from a single template. This is particularly useful for the creation of libraries.

benefits by reducing the number of different types of compilers needed. It also promotes familiarity by programmers and maintenance programmers. Improved program clarity and readability should also simplify the documentation. [Ref. 46]

F. THE PROGRAM LISTING

If the overall goal is to reduce the quantity and cost of documentation and improve its quality and usefulness to the maintenance programmer, then it is highly desirable that programs be made as self-documenting as possible. In this manner one can substantially reduce the necessity to maintain multiple forms of documentation representing the same logic. Many authors advocate such an approach through structured, well commented program listings. Myers [Ref. 18] states:

> Since we already have the code, why not let it serve as the logic documentation?... additional documentation such as a flowchart would be undesirable because it would be redundent with the code. Redundancy in any type of documentation should be avoided because it increases the chances of conflicts. Furthermore, unless care is taken to update the documentation (which is more difficult if the logic documentation is physically seperated from the code), redundent documentation often becomes totally useless after the code is modified a few times.

Glass [Ref. 43] also agrees with this point of view stating:

> The documents, when they do exist, are generally written to conform to a seperate set of requirements which specify what the software documentation is to contain. All too frequently, these requirements provide for irrelevent or useless information that the maintainer really needs. So, in a real sense, the document, which is supposed to be a clarifying piece of material, ends up obscuring the needed information.
>
> Because documentation is seperate from the software product itself, it is also frequently out of date. Ideally, the document would be a perfect reflection of the program. In actual fact, this is rarely, if ever, true. The documentation can therfore be misleading. Who in their right mind would attempt to make corrections to a program after reading only the program documentation and not the listing?

> This author recommends the heresy that the listing be the place where most software documentation is placed. Nearly every requirement for documentation describing a program can be met and in fact exceeded by requiring the same information in the listing.

DOD, in general, supports this point of view and gives explicit direction for what constitutes a well commented program listing in MIL-STD-1679. However, MIL-STD-1679, SECNAVINST 3560.1 and other directives require extensive, detailed, external documentation (i.e., other than the program listing) to be produced. One valid reason for this requirement is the need for extensive design reviews required by configuration management techniques. A second reason is that, until the mid-1970's and the advent of sofware engineering techniques as discussed in this chapter, the program listing did not convey a great deal of information. A large amount of English text was needed to explain the design and structure of the software and its associated data bases. Even Glass admits that some external documentation will always be required to give an overview of the structure and the software's design history [Ref. 43].

This "external" documentation is aimed at specific users. Operator's Manuals are meant to be read by operators. Specifications are meant to be read by both customers (to give them the ability to determine that the problem being solved is the one they want solved) and designers. Test documents are meant to be read by customers, to determine that proper reliability techniques are employed, and testers. The main problem associated with external documentation is that it frequently becomes inaccurate and outdated as maintenance programmers make changes to the program code. For this reason maintenance programmers tend to relay more and more on the program listing as the software system gets older.

> The listing is, of necessity, accurate, since it is the program in all real senses of the word. For the same reason, it is complete. Thus the only accurate and current representation of a program, in today's technology, is frequently the program listing...If the code is changed, it is much more likely that the documentation will be also. In addition, the explanations in the listing will also likely to be readable by the intended target audience, a programmer. They will also be in place where he needs most to find them. The accuracy and completeness attributes of the listing will also tend to apply as well to the documentation. [Ref. 43]

The main line of thought being developed is that recent trends in programming technology, as presented in this chapter, have shifted the emphasis of programming documentation from "external" documents to the program listing itself. A greater detail and quantity of information about the program is now directly available to the maintenance programmer. This has a significant impact on the way one should design documentation as a whole. Chapter IV will discuss this point of view as it applies to a specific document, the Programmer's Maintenance Manual, and how it should be designed.

1. Commenting

It has been stated that one objective to improve the documentation of software was to make the program code itself more readable. Two techniques already discussed were structured programming and the use of high order languages. A third technique is the placing of comments at appropiate places within the program code. One of the main advantages of a consistant commenting policy is its independence from the programming language used. There are very few compilers that do not allow comments to be placed in the listing [Ref. 40].

A second advantage is that commenting closes the gap between computing managers and computer programmers. This gap develops because of the programmer's absolute need for

attention to details. These details include such items as assembly language instruction choices, high order language statement type choices, flag initialization procedures, and the design of nested loops with if..then constructs which implement the logic of the software system. The manager, on the other hand, must keep a "Big-Picture" perspective of the system and be able to evaluate an elusive entity called software quality. Software quality is often based on items such as reliabilty, readability and portablity [Ref. 38]. In order to evaluate all these "-ilities", the manager must be able to read the program listing and understand the implementation of the software design without necessarily being familiar with all the in's and out's of a particular program language.

Comments assist in putting more documentation into the program listing as well as making the program more readible. Comments explain details about the program that are not appearent from the code itself. Table IV provides a list of where comments should appear in the program. One DOD activity, the Marine Corp's Tactical Systems Support Activity, has had a great deal of success in easing its burden of software maintenance by implementing a detailed commenting policy. Although MCTSSA's programs are mainly written in the Navy's CMS-2 language, it was found that such a policy helped reduce the amount of time spent by programmers on software changes in addition to the time savings achieved by the use of a high order language. [Ref. 48]

TABLE IV

Recommended Locations for Comments

1. At the beginnning of each module, include the module name, the current date, the module's function, its inputs and outputs, its limitations and restrictions, including assumptions, its error processes, and the name of the developer. Major modules should also include the history of modifications: for each change, the date, the maintainer's name, purpose of the change and scope.

2. At each subfunction, whether it be a straight sequence of code or a logic branch or a begin-end block, an explanation of that subfunction.

3. At each interface, a clear definition of the interface and a reference for further information about the other side of the interface (where possible).

4. At each group of functionally or otherwise related declarations, an explanation of the role and makeup of the group.

5. At each declaration, an explanation of the role of the item and the meaning, if any, of its possible values.

6. At each difficult-to-understand program portion, an explanation of what the code does and why a complex solution was necessary.

IV. THE MAINTENANCE MANUAL

A. OBJECTIVES OF A MAINTENANCE MANUAL

In order to determine how a programmer's maintenance manual should be organized it will be helpful to examine some specific goals that apply to any type of manual.

1. General

The first objective is to enumerate those general organizational qualities of writing and style that lead to ease of use and readibility of technical publications and documents. There are several factors that insure the information contained in a manual are is easy to use. These factors can be characterized into two broad areas. The first area concerns the concept that all information presented by the manual be easy to find or locate. The second area is the concept that, once one finds the information, the information is readily understood.

The factors that support ease in finding information are consistency, pointers and arrangement [Ref. 50]. Consistency is the principle that similar objects (i.e. maintenance manuals) containing the same information (how to understand and change a computer program) be presented in identical ways. In other words, all manuals should have identical formats. Readers of the manual know what to expect, how to look for specific information and where they will find it. For a program maintenance manual it would be helpful that details on data structures be in the same location in each section of the manual.

Pointers are essential signposts which identify related groups of information. Pointers are represented by entities such as tables of contents, indexes and section headings of text. Pointers announce the presence and location of information within the body of the manual.

Arrangement refers to the manner of presentation used throughout the manual. The arrangement anticipates ways in which readers might look for specific information. The subject of a manual might typically be arranged by alphebetical or chronological order. Subject classification is another method. For a program maintenance manual, the arrangement might be related to the hierarchical design of the main functions and subfunctions of the program or to some external criteria. This criteria could be specified by documentation standards incorporated in directives.

The factors that support ease of understanding are simplicity, concreteness and naturalness [Ref. 50]. Simplicity is the concept that a writer should use a vocabulary and writing style that suits the intended readers. Admittedly, one assumes that any particular person having the need to read a maintenance manual can understand fairly complex compositions. Still, the goal of simplicity is to keep the technical "verbage" to a minimum, while presenting a clear and logical flow of descriptive information. In addition, a dictionary or appendix should be included to supply definitions of any "buzzwords" for clarification and consistancy.

Concreteness ensures that verbal descriptions are more specific than general. It also implies that examples, diagrams and pictures be provided for amplification and clarification. For program maintenance manuals the best type of diagrams to use has seen a long history of controversy. Hierarchy diagrams, flow charts, HIPO charts, and NASSI-SCNEIDERMAN charts, among others, have seen the most

usage [Ref. 51]. New developments such as structured program design languages (PDL's), automated graphic packages and abstract data typing for the design process are now being used to supplement verbal descriptions of the software [Ref. 49].

The concept of naturalness provides the reader with the unfolding of information in an ordered manner. It ensures that readers can verify they are on the right track and will ultimately find what they are looking for [Ref. 52].

2. Record of Design

The second objective of the programmer's maintenance manual is to provide programmers and management a record of the philosophy of design and historical development of the software. The manual must include a concrete representation of the software design as well. Glass [Ref. 43] describes four catagories of documents which could be included as part of a maintenance manual.

- DESIGN NOTES

Design notes explain how and why sections of the software evolved into their present state. They should be prepared in some sort of standard format, filed chronologically and cross-indexed to the program code. They do not have to be detailed but should provide a good overview of the concepts supporting a particular design approach.

- PROBLEM REPORTS

Problem reports are records of problems encountered during the design process. They should describe the problem and its ultimate solution. They are eventually placed in a permanent historic file once the final design is fixed. They are extremely useful in isolating errors that occur during system testing.

- IMPROVEMENT REPORTS

Improvement reports are suggestions and collections of ideas held for future changes to be incorporated into the software system. They can be major improvements or cosmetic only. A reason for doing this is to pass sound ideas of the designers to the maintenance programmers. These ideas may be helpful when adding enhancements as a result of changes in user's requirements.

- VERSION DESCRIPTION

Version descriptions are numbered changes that describe the modifications and enhancements added to a new release of the software. Each numbered change should have a complete list of the changes, where they were made and why, a list of the problem reports closed and a description of the impact of the changes on the users in the user's terms.

The structure and description of the software system design is best kept as simple and straightforward as possible. Before the advent of structured programming and structured design tools (such as Program Design Languages) there was no standardized manner to represent the software design. This is still a problem today, especially when one desires to have "procfs of correctness" to determine that the software is free of errors [Ref. 49]. The best that can be done is to use a combination of items such as hierarchical block diagrams to list the major modules and their control/data flow relationships, grouping of data item descriptions into "clusters" based on usage of the data, and coding the design with a well organized, modularized and readable high order language.

3. Support Maintenance Programmer's Tasks

The programmer's maintenance manual should be the single document that programmers need to refer to for software maintenance activities. The manual has to be complete in that it must contain all the information needed by programmers to accomplish their two primary tasks; correcting errors (modification) and adding new capabilities (enhancement).

To support these tasks the manual should be a well organized, concise and thorough description of the software. It must contain both an overall design view and a discrete, detailed procedure by procedure view. It has to describe all all data items inputed to the program (type/format/range), the processes performed on the data, and the type/format/

range of the output data. In a real-time control system where the software performs control functions based on the parellel processing of data, the logical control responses to various data inputs must be specified.

Data structure and organization has to be described in detail. Enumeration of data names, descriptions of tables and arrays and how they are used, initialized or otherwise manipulated by the program is of primary importance. Cross-reference listings, which list each data item and every module, function and procedure where that data item is used, are valuable aides for understanding the program.

Finally, two general guidelines must be kept in mind if the manual is to support the tasks of the maintenance programmer. First, The manual must provide for complete tracability from the user's operational requirements to the actual program listing (lines of code) so that if a requirement changes then the appropiate code can be correctly changed, deleted or new code added. Second, the manual must be easily modified and the change recorded properly in order to reflect the changes to the software. If this is not done the manual soon becomes outdated and useless as a maintenance tool. [Ref. 43, 51, 53]

B. DEPARTMENT OF DEFENSE REQUIREMENTS

How the objectives of a maintenance manual are best fulfilled is the main topic of this thesis. DoD currently requires a copy of the program listing and several supporting documents for representing the program design. Different DoD agencies have different requirements for documentation and various names for individual documents. The DoD documents briefly described here are from the standard that describes a Program Maintenance Manual for all DoD

activities (DOD STANDARD 7935.1S) and those U.S. Navy standards relating to software maintenance documentation.

1. Program Maintenance Manual

A description of the DoD requirements for a program maintenance manual is presented in DOD STANDARD 7935.1S, "Automated Data Systems Documentation Standards," 13 September 1977. The standards main orientation is toward documenting large data processing systems, however, it can be used for imbedded tactical control systems as well. A copy of the format for the Program Maintenance Manual is provided in Appendix A.

According to the standard, the Maintenance Manual (PMM) is to be divided into four major sections. The first section is required to give a general description of the program; its purpose, history of development, and define other documents used to support the program (User's Manual, etc.). The second section contains a system description to include applications, functions, input/output and information on summary reports. Details and characteristics of each procedure and subroutine that would be of help to the maintenance programmer are to be stated. Items such as data record types, table characterisitcs, exit and branching procedures, interfaces, descriptions of working and output files must be specified. The third section is to describe the operating environment to include what support software is needed. This section is also to contain a complete description of the data base as well as specifying the storage media for the data base (tape, disk, or internal). The fourth section is to contain information on specific maintenance procedures. This will include information on the labeling of functions, subroutines and data records, along with the programming standards utilized. This section is to contain a copy of the program listing itself.

2. Program Description Document

The Program Description Document (PDD) is required by SECNAVINST 3560.1. Its purpose is to provide a complete technical description of all procedures, functions, data structures, operating environment, operating constraints, and data base organization of the software system. The PDD is to describe and completely define the basic program logic and program procedures for each system control subroutine. The PDD is also required to be directly related to the program design specifications which are the formal functional requirements of the software system. The PDD is, in essence, the Navy's version of a DoD program maintenance manual. The PDD does not contain a copy of the program listing or a complete data base description.

3. Data Base Design Document

The Data Base Design Document (DBDD) is required by the same instruction to provide a complete detailed description of all common data items necessary to carry out the functions of the software system. Common data is defined as that data required and used by two or more subprograms. Examples of common data include constants, indexes, flags, variables and tables. The DBDD is to be based on the functional or performance specifications. All terminology in the DBDD is to conform to the programming guidelines of the program design specification and the particular programming language employed. The DBDD has to give an organized, detailed description of all data objects to include such characterisitcs as purpose of the data object, field name, size, numeric type (fixed point, floating point), range of values, initialized condition. A cross-reference listing of each common data item (table, flag, etc.) to each program or subprogram where it is used or set will be provided.

4. Program Package Document

The Program Package Document (PPD) is designed to consist of all the program material items such as card decks, magnetic tape, disk packs or printed listing of the software instructions. The PPD is to include an error free listing of a compilation of the source program and any data which is necessary for the program to run properly. Examples of these data items would be adaption data, data file constants, set-up (initialization) data and program parameter values.

5. Problems with DOD's Requirements

The DoD approach, as standardized in DoD STANDARD 7935.1S, SECNAVINST 3560.1, and MIL-STD-1679(Navy) are all based on the supposedly "good management practice" of having the maintenance documentation of a software system be a complete set of English text. Information concerning the program is compiled into volumes that, in complex systems, can reach several feet in width. All this text information, as briefly outlined previously, gives a system overview, explains each item and structure in the system's data base, shows control flow, data flow, module interfaces, and major functions. All this information, in and of itself, is usable by the maintenance programmer. However, placing it in seperate volumes is not the best way to present it. This information is much more valuable to the programmers when integrated into the program listing. To reiterate: "... Dcoumentation information about a software system belongs, in most cases, in the listing of the program itself." [Ref. 53] There are three key reasons why documentation, to the greatest extent possible, should be placed in the program listing.

The first reason is that programmers tend to dislike writing documentation. They would much rather be writing code, which is what they do best and feel the most comfortable with [Ref. 54]. If the documentation is an integral part of the listing then using readable data names, jotting down a few lines of comment to explain how a procedure or function works and structuring the code becomes a much easier task. The programmer is saved from the tedious paper-work drill of having to look up a separate program maintenance document, figure out how it's organized and where the needed information is and then submit a change outlining what program modifications or enhancements were made. The programmer can be more productive by combining two steps into one by keeping both the documentation and program code up to date. Having the code and documentation together can be used by managers as a motivation factor by demonstrating less work for the programmers in the long run. Other benefits can be shown as in the case of using a programming team to develop a large software project. Here the team can design the documentation as they design the structure of the software. This can eliminate the need for a seperate team just to design and write the documentation. The documentation design can be directly related to and supportive of the software design.

The second reason for placing documentation in the listing is to enhance managements span of control. As mentioned earlier, there is a large requirement for managers to keep a big-picture view and be able to supervise, direct and track programmer's activities and progress while they are performing maintenance tasks. The documentation in the listing provides the means whereby a manager can evaluate changes to the software and its resultant quality. Managers would no longer be required to evaluate changes to the code and it's associated documentation and then have to check to

make sure the documentation change correctly reflects the code change. In essence it is desired to avoid a "double-entry bookkeeping" system where the documentation describes the software as the managers and programmers think it works, and the listing represents how the software actually works. [Ref. 53]

The final reason is that a program maintenance manual must remain accurate and valid as long as the software system is useful. Programmers must be able to quickly logically use the documentation to understand what a section of code is accomplishing and how it is doing so. Again, if user's requirements change, it is essential that the software be changed rapidly and in an error-free manner as possible. If a maintenance programmer cannot tell how the code is working, changes based on valid user needs, or for any other reason, will be difficult at best.

C. A PROPOSED "IDEAL" MAINTENANCE MANUAL

To overcome the difficulties with updating the documentation and ensure that the documentation is in a form that is readily usable by programmers and managers forces one to consider a revised format for a program maintenance manual. The manual contents presented here are based on the advantages inherent with the quality and detail that the DoD standards require and those advantages gained from incorporating current software engineering practices. The "Ideal" program maintnenace manual will be divided into four sections: (1) Overall Program Structure, (2) The Program Listing, (3) A Data Dictionary, and (4) Supplemental Appendices.

1. Overall Program Structure

The overall program structure should consist of words and pictures indicating how the entire system hangs together. A functional block diagram, which shows all major modules, procedures and functions is essential. This diagram represents the system structure, the execution order (if possible) and data flow. The section should contain a well organized index, logically arranged, which points the way to detail level documentation in the listing. The index should reflect the block diagram and the design of the system. The index should locate major data structures and data clusters within each module. This section should contain a written text introduction to the overall purpose and function of the software, the hardware configuration used for tests and evaluation prior to software delivery, and the target computer system (tactical or data processing) the software is to run on. Each module of the program will be listed, a brief description of the module given and the functional relationships/interfaces with other modules completely annotated.

Finally, the section should state the company responsible for the program design and development, the names of the chief programmer and members of his team and applicable references and standards used. These standards should include the program performance specifications, standards for data objects, and the language programming standards.

2. The Program Listing

The computer program listing is the single most important tool for software maintenance activities. The objective of the maintenance manual is to maximize the maintainability aspect of the listing. In this regard clarity

and readibility are to be emphasized over efficiency. It is important that the program listing be clear, concise, structured, well designed, modularized and straightforward. [Ref. 48] Each module should contain a description of what the module does and what procedures or functions are contained in the module. Each module should contain a section for data descriptions or declarations, global and local. Table, variable and flag declarations may be segregated and logically grouped.

a. Physical Layout

Good physical layout is defined as "...that property of a program listing which makes it capable of being read and understood by a programmer not familiar with the program." [Ref. 48] Good readibility may be achieved by a variety of techniques, some of which are; seperation of logically related groups of code, seperation comments and code, blocking (by using lines of asterisks) lengthy comments, the appropiate use of blank lines, logical indentation and the lining up of begin-end, if-then/else pairs.

All the tenants of structured programming , as discussed in Chapter III, are key ingredients of good physical layout. Figures 3.3 and 3.5 illustrate this physical structure. It may be imposed on the code, as with assembly language, or be part of the language syntax, as with ADA. [Ref. 38, 39, 42]

b. Commenting and Naming

The use of meaningful comments is of primary importance to increase understanding of the program. Comments should explain, amplify and supplement the listing rather then echo the code. For example:

```
N = N + 1    --Increment N
```

```
-- A message has just been inserted into the message
-- queue.
-- Increment Msg_QUEUE_Pointer so that it points to
-- the location where the next message may be
-- inserted.

Msg_QUEUE_Pointer = Msg_Queue_Pointer + 1
```

Figure 4.1 An Example of Meaningful Comments.

does nothing to explain why N is being incremented. A better example is illustrated by Figure 4.1 .

If a program module consists of more than one procedure or function then there should be commentary for each procedure or function. The comments for each procedure and function should contain an extensive, detailed description of how the procedure operates and its purpose in the module. The sequence of processing should be described in chronological order to include the calling sequence of control. The hierarchical structure of the module can be reflected in a like manner as comments follow the physical indentation. Table III lists other criteria for commenting as discussed earlier.

All names for data objects, modules and procedures must be descriptive in nature. They should attempt to embody the characterisitcs of the data item they represent. Names such as ID_Buffer, SINE_Function and PAY_Roll_SUM have inherent meanings and are easier for the programmer to trace through the listing. Names such as A, X, N, or XYZ are meaningless. Related data items and procedures should have related names which demonstrate their interconnections. [Ref. 43, 44, 48]

c. Data Declarations and Definitions

All data items should be grouped and organized according to their logical usage. Global data elements should be defined in one location. Local data elements are to described in the procedure where they are manipulated. The technique of information hiding, where the structure and characteristics of local data and parameters are unknown to procedures in other modules [Ref. 24], is to be utilized to the maximum extent.

If the programming language does not support strong data typing for data declarations, as in the DoD high order language ADA, then variable, table, array and other data declarations must contain meaningful comments. These comments are to describe the purpose, initial value, range and distinct structure of each data element. Figure 4.2 reveals how a data table (or record) would be declared in

```
type MONTH_NAME is (JAN,FEB,MAR,APR,MAY,JUN,
                    JUL,AUG,SEP,OCT,NOV,DEC);

type DATE is
     record
        DAY: INTEGER range 1..31;
        MONTH: MONTH_NAME
        YEAR: INTEGER
     end record;
```

Figure 4.2 Example of an ADA Data Table (Record).

ADA. The table is called 'DATE' and has three components named 'DAY', 'MONTH', and 'YEAR'. DAY and YEAR are defined to be of type INTEGER. MONTH has a seperate type declaration called MONTH_NAME. INTEGER is assumed to be defined by

the support software of the system and having the mathematical characteristics of all integer numbers. Variables and constants associated with the table DATE can then be

```
Comment
TABLE ACCOUNTS
        -- Used to store information on 400 Bank
           accounts. Consists of four elements.

        1. ACCOUNT NAME(ACCTNAME)
              -- Contains up to 40 ASCII characters.

        2. ACCOUNT NUMBER(ACCTNR)
              -- Ranges form Zero to 9999 and is of
                 numeric type INTEGER.

        3. BALANCE(BALANCE)
              -- Ranges form -9999.99 to 9999.99
                 dollars and is of numeric type
                 REAL.

        4. FLAG(ACTIVE)
              -- When TRUE (=1), Account is in use.
                 When False (=0), Account is not in
                 use. Is of logical type BOOLEAN.

Miscellaneous
        -- At program initialization time the
           entire table is flushed (set to ZERO).
           Indices (or pointers) related to this
           table are the named variables LASTACCT,
           NEXTACCT and NEWACCT.                    $

TABLE ACCOUNTS V DENSE 400 $
   FIELD ACCTNAME H 20     $
   FIELD ACCTNR   I 14     $
   FIELD BALANCE  A 22 S 7 $
   FIELD ACTIVE   B        $
END-TABLE ACCOUNTS
```

Figure 4.3 Example of a CMS-2 Data Table.

defined. An example would be a variable named 'D' belonging to type DATE. Before D is used or initialized it must be further defined. This is done by by denoting D with a dot followed by the component name as in: (D.DAY:= 4;), (D.MONTH:= JUL;), and (D.YEAR:= 1776). It is clear that all

the attributes of the table called DATE are readily apparent to the maintenance programmer. The readibility of each data object clarifies the purpose and structure of the table. [Ref. 44] Figure 4.3 gives an example of a common way to declare a table using the Navy's CMS-2 programming language. The use of clear and concise comments fulfills similar objectives as data typing in ADA.

3. A Data Dictionary

A data dictionary provides descriptions of individual data entities and can be used as an index as to where the data elements are declared in the program listing. This is extremely useful for large programs, i.e., greater than 10,000 lines of code. The data dictionary can and should provide the formats for the declaration of data within the program while being a catalog of the data resources of a software system. [Ref. 55]

The data dictionary defines the logical organization and physical organization of the system's data entities. The logical organization specifies requirements for data access, modification, associativity and other system orientated concerns. The physical organization defines file structures, record formats, hardware dependent processing features and database management characteristics. All data structures and the operations that are to be performed on each structure should be identified in the dictionary. The program listing can be referenced for details on data elements and those functions or operations dependent on these elements. A data dictionary can explicitly represent the relationships among data and the constraints these elements place on data structures. Algorithms that may take advantage of specific data relationships can be more easily designed and modified if a dictionary-like data specification exists. [Ref. 56]

It is appropiate that the data dictionary reflect the structure of the program listing as closely as possible. The concept of "mapping" the dictionary onto the listing ensures the consistancy required by maintenance actions. It is critical to the maintenance programmer to know which data items effect which particular modules and vice versa. A carefully designed and integrated data dictionary is an essential tool for any modifications or enhancements to the software.

4. Appendices

This section of the maintenance manual is to contain that supplementary information which is of a historical nature. Examples would be notes on the development of the software design, problem reports and improvement reports as described by Glass [Ref. 43]. In a seperate appendix could be a complete description of the intended operating environment, hardware configuration and operating system support desired or required. A continuous file of each version release and a log of all changes made to the program (who made them and for what reason) should be included. [Ref. 40, 42, 48]

A final appendix should contain a set of standards for commenting, algorithmic structures, and the data dictionary. The standards for commenting are useful for consistancy and to enforce readibility of the listing. Standard algorithmic structures provide a framework for the development of module libraries. Such libraries can be utilized to add enhancements to the program in a short period of time since the modules involved are already tested for error free operation and the functions or services provided by the module are well known. Standards for the data dictionary provide common interfaces between the hardware/software system and the user organization. This is

accomplished by specifying the flow and storage locations of data entities within the organization or within the computer installation [Ref. 55]. Standards for a data dictionary can also provide a library of standardized data structure templates to be used for representing the logical and physical characteristics of data entities within the software system database [Ref. 56].

V. CONCLUSIONS AND RECOMMENDATIONS

The Department of Defense has an urgent requirement to reduce the costs of software maintenance during the coming decade. Recent advancements in the methods of software engineering such as modularization, structured design, structured programming and data abstraction have contributed to greater program comprehension. This increased comprehension leads to easier modifications to meet a dynamically changing environment.

It is the opinion of this author that the benefits obtained from proven software engineering practices can be realized in the program maintenance manual. These principles can and should be applied to the design and standards for such a manual. The information available strickly from the program code itself forces us to question the practice of keeping the documentation seperate from the code, and leads to the conclusion that it should not be seperate but integrated into the listing.

With this in mind the following recommendations are put forth:

- The present standards for software documentation be revised to incorporate the most useful aspects of recent software engineering technology.
- Studies should be undertaken with the goal of standardizing terminology for software maintenance documents among all DoD activities.
- A framework has been proposed for a program maintenance manual. This framework should be implemented and tested on various size and types of software systems to discover what savings in time and money can be achieved.

APPENDIX A

PROGRAM MAINTENANCE MANUAL (DOD)

from: DOD STANDARD 7935.1S, "Automated Data Systems Documentation Standards," 13 September 1977

PROGRAM MAINTENANCE MANUAL
TABLE OF CONTENTS

SECTION 1. GENERAL DESCRIPTION

1.1 Purpose of the Program Maintenance Manual. This paragraph shall describe the purpose of the MM (Program Maintenance Manual) in the following words or appropiate modifications thereto:

> The objective for writing this Program Maintenance Manual for (Project Name) (Project Number) is to provide the maintenance programmer personnel with the information necessary to effectively maintain the system.

1.2 Project References. This paragraph shall provide a brief summary of the references applicable to the history and development of the project. The general nature of the system (tactical, inventory control, war-gaming, management information, etc.) developed shall be specified. A brief description of this system shall include its purpose and uses. Also indicated shall be the project sponsor and user as well as the operating center(s) that will run the completed computer programs. A list of applicable documents shall be included. At least the following shall be specified, when applicable, by author or source, reference number, title and security classification:

a. Users Manual.

b. Computer Operation Manual.

c. Other pertinent documentation on the project.

1.3 Terms and Abbreviations. This paragraph shall provide a list or include in an appendix any terms, definitions or acronyms unique to this document and subject to interpretation by the user of the document. This list will not include item names or data codes.

SECTION 2. SYSTEM DESCRIPTION

2.1 System Application. The purpose of the system and the functions it performs shall be explained. A particular application system, for example, might serve to control mission activities by accepting specific inputs (status reports, emergency conditions), extracting items of data, and deriving other items of data in order to produce both information about a specific mission and information for summary reports. These functions shall be related to paragraphs 3.1, Specific Performance Requirements, and 3.2, System Functions, of the FD (Functional Description).

2.2 Security and Privacy. This paragraph shall describe the classified components of the system, including inputs, outputs, data bases, and computer programs. It will also prescribe any privacy restrictions associated with the use of the data.

2.3 General Description. This paragraph will provide a comprehensive description of the system, subsystem, jobs, etc. in terms of their overall functions. This description will be accompanied by a chart showing the interrelationships of the major components of the system.

2.4 Program Description. The purpose of this paragraph is to supply details and characteristics of each program and subroutine that would be of value to a maintenance programmer understanding the program and its relationship to other programs. (Special maintenance programs related to the specific system being documented will be discussed under paragraph 4.4, Special Maintenance Procedures.) This paragraph will initially contain a list of all programs to be discussed, followed by a narrative description of each program and its respective subroutines under seperate paragraphs starting with 2.4.1 through 2.4.n. Information to be included in the narrative description is represented by the following items:

a. Identification - program number or tag including a designation of the version number of the program.

b. Functions - description of program functions and method used in the program to accomplish the function.

c. Input - description of the input. Descriptions here must include all information pertinent to maintenance programming, including:
   (1) Data records used by the program during operation
   (2) Input data type and location(s) used by the program when its operation begins.
   (3) Entry requirements concerning the initiation of the program.

d. Processing - description of the processing performed by the program, including:

(1) Major operations - the major operations of program will be described. The description may reference chart(s) which may be included in an appendix. This chart will show the general logical flow of operations, such as read an input, access a data record, major decision, and print an output which would be represented by segments or subprograms within the program. Reference may be made to included charts that present each major operation in more detail.
(2) Major branching conditions provided in the program.
(3) Restrictions that have been designed into the system with respect to the operation of this program, or any limitations on the use of the program.
(4) Exit requirements concerning termination of the operation of the program.
(5) Communications or linkage to the next logical program (operational, control).
(6) Output data type and location(s) produced by the program for use by related processing segments of the system.
(7) Storage - Specify the amount and type of storage required to use the program and the broad parameters of the storage locations needed.

e. Output - description of the outputs produced by the program. While this description may reference output described in the Users Manual, any intermediate output, working files, etc. should be described for the benefit of the maintenance programmer.

f. Interfaces - description of the interfaces to from this program.

g. Tables and Items - provide details and characteristics of the tables and items within each program. Items not part of a table must be listed seperately. Items contained within a table may be referenced from the table descriptions. If the data description of the program provides sufficient information, the program listing may be referenced to provide some of the necessary information. At least

the following will be included for each table:

(1) Table tag, lable or symbolic name.
(2) Full name and purpose of the table.
(3) Other programs that use this table.
(4) Logical divisions within the table (internal table blocks or parts - not entries).
(5) Basic table structure (fixed or variable length, fixed or variable entry structure).
(6) Table layout (a graphic presentation should be used). Included in supporting description should be table controlling information, details of the structure of each type of entry, unique or significant characteristics of the use of the table, and information about the names and locations of items within the table.
(7) Items - the term "item" refers to a specific category of detailed information that is coded for direct and immediate manipulation by a program. Used in this sense, the definition of an item is machine- and program-oriented rather than operationally oriented. Of primary importance is an explanation of the use of each item. At least the following will be included for each item:

(a) Item tag or label and full name.
(b) Purpose of the item.
(c) Item coding, depending upon the item type, such as integer, symbolic, status, etc.

h. Unique Run Features - description of any unique features of the running of this program that are not included in the Computer Operation Manual.

SECTION 3. ENVIRONMENT

3.1 Equipment Environment. This paragraph shall discuss the equipment configuration and its general characteristics as they apply to the system.

3.2 Support Software. This paragraph shall list the various support software used by the system and identify the version or release number under which the system was developed.

3.3 Data Base. Information in this paragraph shall include a complete description of the nature and content of each data base used by the system.

3.3.1 General Characteristics. Provide a general description of the characteristics of the data base, including:

a. Identification - name and mnemonic reference of the component (e.g. data base). List the programs utilizing the component and explain the use of the component in the system.

b. Permanency - note whether the component contains static data that a program can reference, but may not change, or dynamic data that can be changed or updated during system operation. Indicate whether the change is periodic or random as a function of input data.

c. Storage - specify the storage media for the data base (e.g. tape, disk, internal storage) and the amount of storage required.

d. Restrictions - explain why limitations on the use of this component by the program in the system.

3.3.2 Organization and Detailed Description. This paragraph will serve to define the internal structure of the data base. A layout will be shown and its composition, such as records and tables, will be explained. If available, computer generated or other listings of this detailed information may be referenced or included herein. The following items indicate the type of information desired:

a. Layout - show the structure of the data base including record and items.

b. Sections - note whether the physical record is a logical record or one of several that constitute a logical record. Identify the record parts, such as header or control segments and the body of the record.

c. Fields - identify each field in the record structure and, if necessary, explain its purpose. Include for each field the following items:

(1) Tags/labels - indicate the tag or label assigned to reference each field.
(2) Size - indicate the length and number of bits/characters that make up each data field.
(3) Range - indicate the range of acceptable values for the field entry.

d. Expansion - note provisions, if any, for adding additional data fields to the record.

SECTION 4. PROGRAM MAINTENANCE PROCEDURES

Section 4 of the manual shall provide information on the specific procedures necessary for the programmer to maintain the programs that make up the system.

4.1 Conventions. This paragraph will explain all rules, schemes, and conventions that have been used within the system. Information of this nature could include the following items.

a. Design of mnemonic identifiers and their application to the tagging or labeling of programs, subroutines, records, data fields, storage areas, etc.

b. Procedures and standards for charts, listings, serialization of cards, abbreviations used in statements and remarks, and symbols appearing in charts and listings.

c. The appropiate standards, fully identified, may be referenced in lieu of a detailed outline of conventions.

d. Standard data elements and related features.

4.2 Verification Procedures. This paragraph will include those requirements and procedures necessary to check the performance of a program section following its modification. Included may also be procedures for the periodic verification of the program.

4.3 Error Conditions. A description of error conditions, not previously documented, may also be included. This description shall include an explanation of the source of the error and recommended methods to correct it.

4.4 Special Maintenance Procedures. This paragraph shall contain any special procedures required which have not been delineated elsewhere in this section. Specific information that may be appropiate for presentation shall include:

a. Requirements, porcedures, and verification which may be necessary to maintain the system input-output components, such as the data base.

b. Requirements, procedures, and verification methods necessary to perform a Library Maintenance System run.

4.5 Special Maintenance Programs. This paragraph shall contain an inventory of any special programs (such as file restoration, purging historical files) used to maintain the system. These programs should be described in the same manner as those described in paragraphs 2.3 and 2.4 of the MM.

a. Input-Output Requirements.
Included in this paragraph shall be the requirements concerning the equipment and materials needed to support the necessary maintenance tasks. Materials may, for example, include card decks for loading a maintenance program and the inputs which represent the changes to be made. When a support system is being used, this paragraph should reference the appropiate manual.

b. Procedures
The procedures, presented in a step-by-step manner, shall detail the method of preparing the inputs, such as structuring and sequencing of inputs. The operations or steps to be followed in setting up, running, and terminating the maintenance task on the equipment shall be given.

4.6 Listings. This paragraph will contain or provide a reference to the location of the program listing. Comments appropiate to particular instructions shall be made if necessary to understand and follow the listing.

LIST OF REFERENCES

1. United States Government Accounting Office Report AFMD-81-25, Federal Agencies' Maintainence of Computer Programs: Expensive and Undermanaged, p. 1, February 1981.

2. Brooks, F.P. Jr., The Mythical Man-Month, Addison-Wessely, 1975.

3. Boehm, B., "Software and Its Impact: A Quantitative Assessment," Datamation, v. 19, no. 5, pp. 48-59, May 1973.

4. Defense Science Board, Report of the Task Force on Technology Based Strategy, p. 41, October 1976.

5. Kline, N. B., and Scheidewind, N.F., "Life Cycle Comparisions of Hardware and Software Maintainability," Third Natoial Reliability Conference, pp. 4a/3/1-14, 1981.

6. Lehman, J. H., "How Software Projects are Really Managed," Datamation, v. 25, pp. 118-129, 1979.

7. Leintz, B.P., Swanson, B.E., and Tompkins, G.E., "Characteristics of Application Software Maintenance," Communications of the ACM, v. 21, no. 6, pp. 466-471, 1978.

8. Munson, John B., "Software Maintainability: A Practical Concern for Lifecycle Costs," COMPSAC, pp. 54-59, 1978.

9. Cheatam, Thomas E., The High Costs of Software, National Technical Information Service, 1973.

10. Lyons, Michael J., "Salvaging Your Software Asset (Tools Based Maintenance)," AFIPS, pp. 337-341, 1981.

11. Mills, H.D., "On the Statistical Validation of Computer Programs," Proceedings of the Second International Conference on Software Engineering, October 13-15, pp. 28-37, 1975.

12. Reutter, John, III, "Maintenance is a Management Problem and a Programmer's Opportunity," AFIPS, pp. 343-347, 1981.

13. DeRoze, B., and Nyman, T., "The Software Life Cycle - A Management and Technological Challange in the Department of Defense," IEEE Transactions on Software Engineering, SE-4, no. 4, pp. 309-318, July 1978.

14. Yau, Stephen S., and Collofello, James S., "Some Stability Measures for Software Maintenance," IEEE Transactions on Software Engineering, v. 6, no. 6, pp. 545-552, 1980.

15. Naval Postgraduate School Technical Report NPS-54-82-002, Software Maintenance Improvement Through Better Development Standards and Documentation, by Schneidewind, N.F., Naval Postgraduate School, Monterey, CA, pp. 23-26, February 1982.

16. Pilcher, Russell D., Techniques Available For Improving the Maintainability of DOD Weapon System Software, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1980, p. 9-10.

17. DeRoze, B. C., Special Presentation, Proceedings on Managing the Development of Weapons Systems Software Conference, pp. 4-2 - 4-12, May 1975.

18. Myers, G. J., Software Reliability Principles and Practices, John Wiley & Sons, 1976.

19. Leintz, B.P., and Swanson, E.B., Software Maintenance Management, Addison-Wesley, Reading Ma., 1980.

20. Silbey, Valdur, "Documentation Standardization," Data Management, v. 17, no. 4, pp. 32-35, April 1979.

21. National Bureau of Standards Report NBS-SP-500-11 Computer Software Management - A Primer For Project Management and Quality Control, by D.W. Fife, pp. 1-10, July 1977.

22. Boehm, D. W., "Software Engineering," IEEE Transactions on Computers, v. C-25, No. 12, December 1976.

23. Riche, Robert S., and Williams, Charles E., A Software Foundation For AN/SPY-1A Radar Control, Masters Thesis, Naval Postgraduate School, Monterey, California, pp. 72-73, December 1981.

24. Parnas, David, L., "Design Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, p. 226-230, March 1979.

25. Freeman, P., and Wasserman, A.I., Tutorial on Software Design Techniques, 3rd Edition, IEEE, New York, N.Y., p. 2-4, 1980.

26. Tauseworthe, R.C., Standardized Development of Computer Software, (Part I, Methods Part II, Standards), Jet Propulsion Laboratory, California Institute of Technology, Part I: 1976, Part II: 1978.

27. Swanson, E.B., "The Dimensions of Maintenance," Proceedings 2nd International Conference on Software Engineering, pp. 492-497, 1975.

28. Rome Air Development Center Report RADC-TR-79-200, Reliability and Maintainability Management Manual, by A. Coppola and A.N. Sukert, pp. 127-151, July 1979.

29. Cave, W.C., and Salisbury, A.B., "Controlling the Software Life Cycle - The Project Management Task," IEEE Transactions on Software Engineering, pp. 326-334, July 1978.

30. Cooper, J.D., "Corporate Level Software Management," IEEE Transactions on Software Engineering, pp. 319-325, July 1978.

31. Daly, E.B., "Management of Software Development," IEEE Transactions on Software Engineering, pp. 229-242, May 1977.

32. MITRE Corporation, DOD Weapons Systems Software Acquisition and Management Study, MTR-6908, v. I, June 1975, (DLSIE Accession No: LD 38652A).

33. Applied Physics Laboratory, The John Hopkins University, Report SR-75-3, DOD Weapon System Software Management Study, by Kossiakoff, A., et al., June 1975 (DTIC Accession No: AD-A022160).

34. Assistant Secretary of Defense, Defense System Software Management Plan, March 1975, (DTIC Accession No: AD-A022558).

35. Stanfield, J.R., and Skrukrud, A.M., Software Acquisition Management Guidebook, Software Maintenance Volume, Systems Development Corp., TM-5772/004/02, November 1977, (DTIC Accession No: AD-A053040).

36. Bersoff, E.H., Henderson, V.D., and Siegel, S.G., "Software Configuration Management: A Tutorial," Computer, pp. 6-14, January 1979.

37. Naval Postgraduate School Technical Report NPS-54-82-003, Evaluation of SECNAVINST 3560.1: Tactical Digital Systems Documentation Standard For Software Maintenance, by Scheidewind, N.F., Naval Postgraduate School, Monterey, CA, pp. 22-24, 1982.

38. Ross, D.T., Goodeneough, J.B., and Irvine, C.A., "Software Engineering: Principles, Processes and Goals," Computer, p. 55, May 1975.

39. Baker, A., "Structured Programming in a Production Programming Environment," Proceedings of the IEEE International Conference on Reliable Software, 1975.

40. Linger, R.C., and Mills, H.D., "On the Development of Large Reliable Programs," Current Trends in Programming Methodology, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

41. Linger, R.C., Mills, H.D., and Witt, B.J., Structured Programming: Theory and Practice, Addison-Wesley Publishing Co., Reading Massachusetts, pp. 15-16, 1979.

42. Wulf, W.A., "Languages and Structured Programs," Current Trends in Programming Methodology, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

43. Glass, R.L., and Noiseux, R.A., Software Maintenance Guidebook, Prentice-Hall, Englewood Cliffs, New Jersey, pp. 84-130, 1981.

44. Barnes, J.G.P., Programming in ADA, Addison-Wesley Publishing Co., Reading, Massachussetts, 1981.

45. Jackson, M.L., Principles of Program Design, Academic Press, p. 30, 1975.

46. "Department of Defense Requirements for the Programming Environment for the Common High Order Language - PEBBLEMAN:Revised," Defense Advanced Research Projects Agency, Arlington, Virginia, January 1979.

47. Ritchie, J., and Thompson B., "The UNIX Time-Sharing System," The Bell System Technical Journal, p. 20, July-August 1978.

48. Marine Corp's Tactical Software Support Activity, "Standards and Conventions for Use of the CMS-2 Language," Camp Pendelton, California, pp. 10-20, 1979.

49. Liskov, B. H., "A Design Methodology for Reliable Software Systems," Proceedings Fall Joint Computer Conference, pp. 65-73, 1972.

50. Bethke, F.J., et. al., "Improving the Usability of Programming Publications," IBM Systems Journal, v. 20, No. 3, pp. 306-320, January 1981.

51. Snyders, J., "User Manuals that Make Sense," Computer Decisions, No. 13, pp. 125-128, April 1981.

52. Judisctt, J.M., Rupp, B.A., and Dassinger, R.A., "Effects of Manual Style on Performance in Education and Machine Maintenance," IBM Systems Journal, v. 20, No. 2, pp. 172-183, February 1981.

53. Wilson, L., "The Do's and Don'ts of Documentation," Datamation, v. 27, pp. 185-186, September 1981.

54. Weinberg, G.M., The Psychology of Computer Programming, Van Nostrand Reinhold Co., p. 50, 1971.

55. Federal Information Processing Standards Publication 76 (FIPS PUB-76), Specifications and guidelines for Planning and Using a Data Dictionary System, United States Department of Commerce, National Bureau of Standards, p. 6, 20 August 1980.

56. Pressman, Roger S., Software Engineering, A Practitioner's Approach, McGraw-Hill Book Company, New York, pp. 228-230, 1982.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 54 Department of Administrative Science Naval Postgraduate School Monterey, California 93940	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Curricular Officer, Code 37 Computer Systems Technology Naval Postgraduate School Monterey, California 93940	1
6.	Professor Norman Lyons, Code 54Lb Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
7.	Professor Roger Weissinger-Baylon, Code 54Wr Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
8.	LCDR Ronald Modes, Code 52Mf Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
9.	Major W.D. Helling, USMC 2511 Windsor Avenue Dubuque, Iowa 52001	1
10.	LT James Howard Teuscher, USN Post Office Box 46 Wells, New York 12190	2

200174

Thesis
T358
c.1

Teuscher
Software engineering practices:
their impact on the design of a program maintenance manual.